第 1 章　使用现代移动图形硬件

本章介绍使用嵌入式图形硬件绘制 3 维（3D）图形的最新方法。嵌入式系统涵盖了范围广泛的设备，从飞机驾驶员座舱显示器到自动售货机。绝大多数具有 3D 功能的嵌入式系统都是手持电脑，比如苹果的 iPhone、iPod Touch、iPad，或者是基于谷歌的 Android 操作系统的手机。索尼、任天堂及其他的手持设备也具备强大的 3D 图形能力。

用于嵌入式系统的 OpenGL（OpenGL ES）定义了嵌入式 3D 图形的标准。基于 iOS 5 的 iPhone、iPod Touch 以及 iPad 设备支持的是 OpenGL ES 2.0。苹果的设备也支持旧的 OpenGL ES 1.1 版本。iOS 5 引入了 GLKit 软件框架，这个框架简化了很多常用的编程任务，同时部分隐藏了所支持的这两个 OpenGL ES 版本间的差异。本书主要关注带有 GLKit 的 iOS 5 所支持的 OpenGL ES 2.0 版本。

为了能在 ANSI C 编程语言中使用，OpenGL ES 定义了一个应用程序编程接口。通常用来开发苹果产品的 C++ 和 Objective-C 编程语言可以与 ANSI C 无缝交互。特定的转换层或者粘合层的存在使 OpenGL ES 可以用在 JavaScript 和 Python 中。新兴的 Web 编程标准，如非营利性 Web3D 联盟的 WebGL 标准，也正准备在网页上实现对于 OpenGL ES API 的跨平台标准化访问。本书中讲解的 3D 图形概念适用于所有具有 3D 功能的嵌入式系统。

本章会讲解使用 OpenGL ES 和 iOS 5 来实现 3D 图形的一般方法，但并不会深入特定的编程细节。最新的 3D 图形硬件加速是所有高级移动产品的可视化效果的基础。学习本章是能够从移动硬件中萃取出最好的 3D 图形和可视效果的第一步。

1.1　3D 渲染

图形处理单元（GPU）就是能够结合几何、颜色、灯光和其他数据而产生一个屏幕图像的硬件组件。屏幕只有 2 维，因此显示 3D 数据的技巧就在于产生能够迷惑眼睛使其看到丢失的第 3 维的一个图像，参见图 1-1 中的例子。

用 3D 数据生成一个 2D 图像的过程就叫渲染。在计算机上显示的图片是由矩形的颜色点组成的，这些矩形的颜色点叫做像素。图 1-2 放大了图像中的一部分来显示

单独的像素。如果通过放大镜仔细观察显示器，你会看到每个像素都是由 3 个颜色元素组成的，即一个红点、一个绿点和一个蓝点。图 1-2 还显示了一个进一步放大的像素来描述单独颜色的元素。在一个全彩色的显示器上，一个像素通常含有红、绿、蓝 3 个元素，但这些元素的排列样式可能与图 1-2 中显示的一个挨一个的排列方式不同。

图 1-1　用 3D 数据生成的一个示例图像

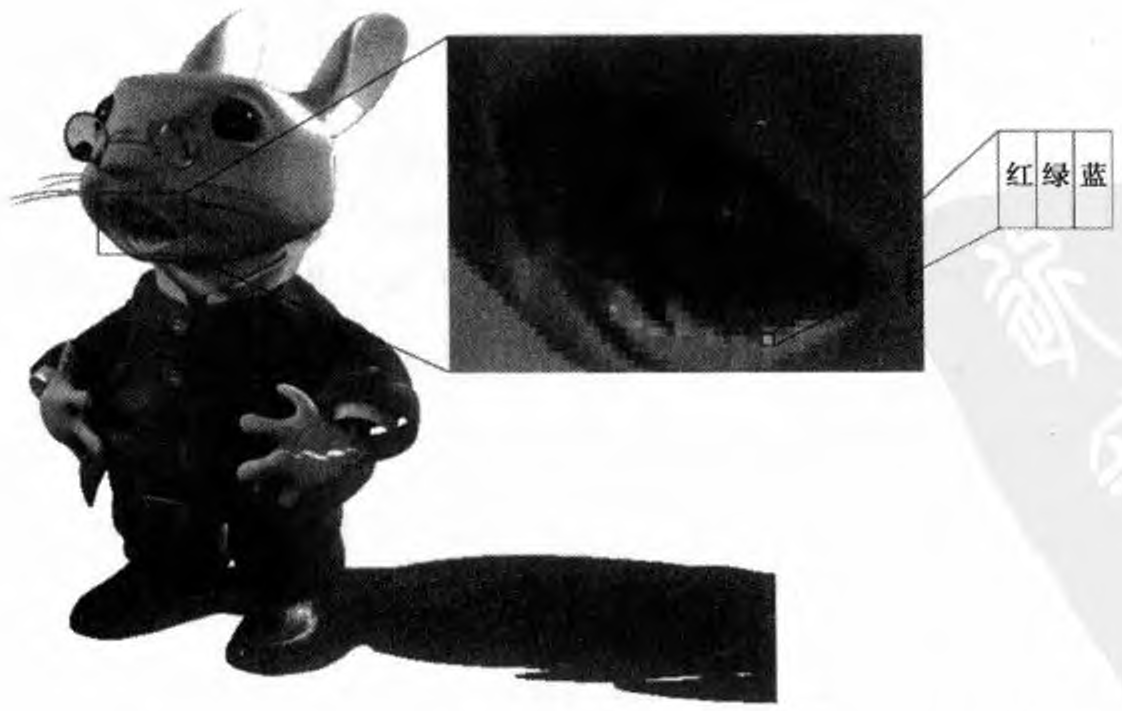


图 1-2　图像是由像素构成的，每个像素有红、绿、蓝 3 个元素

图像是以每个像素至少包含 3 个值的一个数组存储在电脑的存储器中的。第一个值指定了像素的红色元素的强度，第二个值代表绿色强度，第三个值是蓝色强度。一个包含 10000 个像素的图像能够以一个拥有 30000 个强度值的数组的形式存储在存储器中，像素的 3 个元素每个需要一个值。以不同的强度结合红绿蓝 3 个值就足以产生彩虹的所有颜色。如果 3 个元素的强度都是 0，结果颜色就是黑色。如果 3 个颜色都是强度的最大值，结果颜色就是白色。黄色是通过丢掉蓝色并混合红色和绿色得到的。图 1-3 中的 Mac OS X 标准颜色面板用户界面包含能够调节相关的红、绿、蓝强度的图像滑块。

图 1-3　用于调节红、绿、蓝颜色元素强度的用户界面

渲染 3D 数据为一个 2D 图像通常发生在几个不同的步骤中，包括设置图像中的每个像素的红、绿、蓝颜色强度的计算。总的来说，本书会讲解怎么让程序在渲染过程的每一步中尽可能地利用好 OpenGL ES 和图形硬件。第一步是为 GPU 提供要处理的 3D 数据。

1.2　为图形处理器提供数据

程序会保存 3D 场景数据到硬件随机存取存储器（RAM）中。嵌入式系统的中央处理单元有专门为其分配的 RAM。在图形处理的过程中，GPU 也有专门为其分配的 RAM。使用现代硬件渲染 3D 图形的速度几乎完全取决于不同的内存区域被访问的方式。

OpenGL ES 是一种软件技术。OpenGL ES 部分运行在 CPU 上，部分运行在 GPU 上。OpenGL ES 横跨在两个处理器之间，协调两个内存区域之间的数据交换。图 1-4 中的箭头代表了与 3D 渲染相关的硬件组件之间的数据交换。每个箭头也代表着一个渲染性能的瓶颈。OpenGL ES 通常会高效地协调数据交换，但是程序与 OpenGL ES 的交互方式会明显地增加或者减少所需的数据交换的数量和类型。对于渲染速度，最快的数据交换方式是没有数据交换。

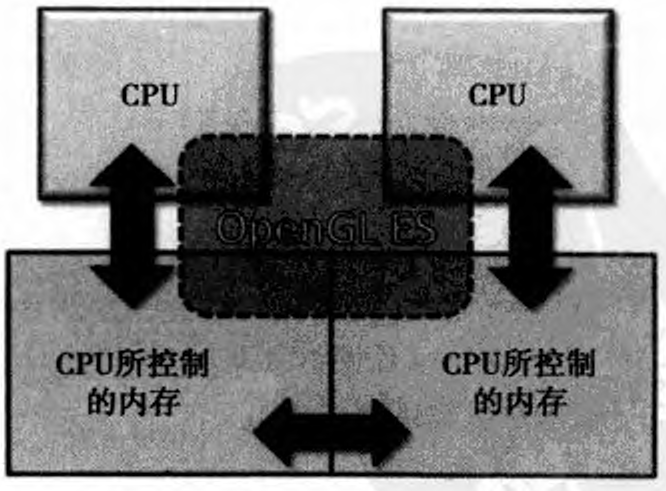


图 1-4　硬件组件和 OpenGL ES 之间的关系

首先，从一个内存区域复制数据到另一

个内存区域速度是相对较慢的。更糟糕的是，除非非常小心，在内存复制发生的时候GPU 和 CPU 都不能把内存另作它用。因此内存区域之间的数据交换需要尽量避免。

其次，所有的内存访问都是相对较慢的。最新的嵌入式 CPU 可以很容易地完成大约每秒一亿次的运算，但是它只能每秒读写内存 200 万次。这意味着，除非 CPU 能够在每次从内存读取一块数据后有效地运行五个或者更多个运算，否则处理器的性能就处于次优状态，这种状态叫做“数据饥饿”。这种情况对于 GPU 来说更明显，在理想条件下，GPU 能够每秒执行数亿次运算，但是却只能每秒访问内存 2 亿次。GPU 几乎总是受限于内存访问的性能，并且通常需要在每块数据上执行 10 ～ 30 次运算才不会影响整体的图形输出。

概括最新 OpenGL ES 和以前的 OpenGL 版本之间差异的一种方式是，最新的 OpenGL ES 为了支持新改进的方法抛弃了对于旧式的低效的内存复制操作的支持。如果你曾做过老式的桌面 OpenGL 开发，那么现在可以忘掉那些经验了。现在的嵌入式系统不再支持以前的糟糕技术了。OpenGL ES 仍然支持多种为图形处理器提供数据的方式，但只存在一种最好的方式，并且本书会始终使用这种方式。

1.2.1 缓存：提供数据的最好方式

OpenGL ES 为两个内存区域间的数据交换定义了缓存（buffers）的概念。缓存是指图形处理器能够控制和管理的连续 RAM。程序从 CPU 的内存复制数据到 OpenGL ES 的缓存。在 GPU 取得一个缓存的所有权以后，运行在 CPU 中的程序理想情况下将不再接触这个缓存。通过控制独占的缓存，GPU 就能够尽可能以最有效的方式读写内存。图形处理器把它处理大量数据的能力异步同时地应用到缓存上，这意味着在 GPU 使用缓存中的数据工作的同时，运行在 CPU 中的程序可以继续执行。

几乎所有程序提供给 GPU 的数据都应该放入缓存中。缓存存储的到底是几何数据、颜色、灯光效果，还是其他信息并不重要。为缓存提供数据有如下 7 个步骤。

1）生成（Generate）——请求 OpenGL ES 为图形处理器控制的缓存生成一个独一无二的标识符。

2）绑定（Bind）——告诉 OpenGL ES 为接下来的运算使用一个缓存。

3）缓存数据（Buffer Data）——让 OpenGL ES 为当前绑定的缓存分配并初始化足够的连续内存（通常是从 CPU 控制的内存复制数据到分配的内存）。

4）启用（Enable）或者禁止（Disable）——告诉 OpenGL ES 在接下来的渲染中是否使用缓存中的数据。

5）设置指针（Set Pointers）——告诉 Open-GL ES 在缓存中的数据的类型和所有需要访问的数据的内存偏移值。

6）绘图（Draw）——告诉 OpenGL ES 使用当前绑定并启用的缓存中的数据渲染整个场景或者某个场景的一部分。

7）删除（Delete）——告诉 OpenGL ES 删除以前生成的缓存并释放相关的资源。

理想情况下，每个生成的缓存都可以使用一个相当长的时间（可能是程序的整个生命周期）。生成、初始化和删除缓存有时需要耗费时间来同步图形处理器和 CPU。存在这个延迟是因为 GPU 在删除一个缓存之前必须完成所有与该缓存相关的等待中的运算。如果一个程序每秒生成和删除缓存数千次，GPU 可能就没有时间来完成任何渲染了。

OpenGL ES 为一种类型的缓存在使用过程中的每一个步骤的执行定义了下面的 C 语言函数，同时为其他类型的缓存提供了类似的函数。

- glGenBuffers()——请求 OpenGL ES 为图形处理器控制的缓存生成一个独一无二的标识符。
- glBindBuffer()——告诉 OpenGL ES 为接下来的运算使用一个缓存。
- glBufferData() 或者 glBufferSubData()——让 OpenGL ES 为当前绑定的缓存分配并初始化足够的连续内存（通常是从 CPU 控制的内存复制数据到分配的内存）。
- glEnableVertexAttribArray() 或者 glDisableVertexAttribArray()——告诉 OpenGL ES 在接下来的渲染中是否使用缓存中的数据。
- glVertexAttribPointer()——告诉 OpenGL ES 在缓存中的数据的类型和所有需要访问的数据的内存偏移值。
- glDrawArrays() 或者 glDrawElements()——告诉 OpenGL ES 使用当前绑定并启用的缓存中的数据渲染整个场景或者某个场景的一部分。
- glDeleteBuffers()——告诉 OpenGL ES 删除以前生成的缓存并释放相关的资源。

注意　这里提到的 C 函数只是为了描述 OpenGL ES 2.0 API 函数与基础概念的对应方式。全书会用各种例子讲解这类 C 函数，因此现在不用担心记不住它们。

1.2.2　帧缓存

GPU 需要知道应该在内存中的哪个位置存储渲染出来的 2D 图像像素数据。就像为 GPU 提供数据的缓存一样，接收渲染结果的缓冲区叫做帧缓存（frame buffer）。程序会像任何其他种类的缓存一样生成、绑定、删除帧缓存。但是帧缓存不需要初始化，因为渲染指令会在适当的时候替换缓存的内容。帧缓存会在被绑定的时候隐式开启，同时 OpenGL ES 会自动地根据特定平台的硬件配置和功能来设置数据的类型和偏移。

可以同时存在很多帧缓存，并且可以通过 OpenGL ES 让 GPU 把渲染结果存储到任意数量的帧缓存中。但是，屏幕显示像素要受到保存在前帧缓存（front frame buffer）的特定帧缓存中的像素颜色元素的控制。程序和操作系统很少会直接渲染到前帧缓存

中，因为那样会让用户看到正在渲染中的还没渲染完成的图像。相反，程序和操作系统会把渲染结果保存到包括后帧缓存（back frame buffer）在内的其他帧缓存中。当渲染后的后帧缓存包含一个完成的图像时，前帧缓存与后帧缓存几乎会瞬间切换。后帧缓存会变成新的前帧缓存，同时旧的前帧缓存会变成后帧缓存。图 1-5 展示了屏幕显示像素、前帧缓存及后帧缓存三者之间的关系。

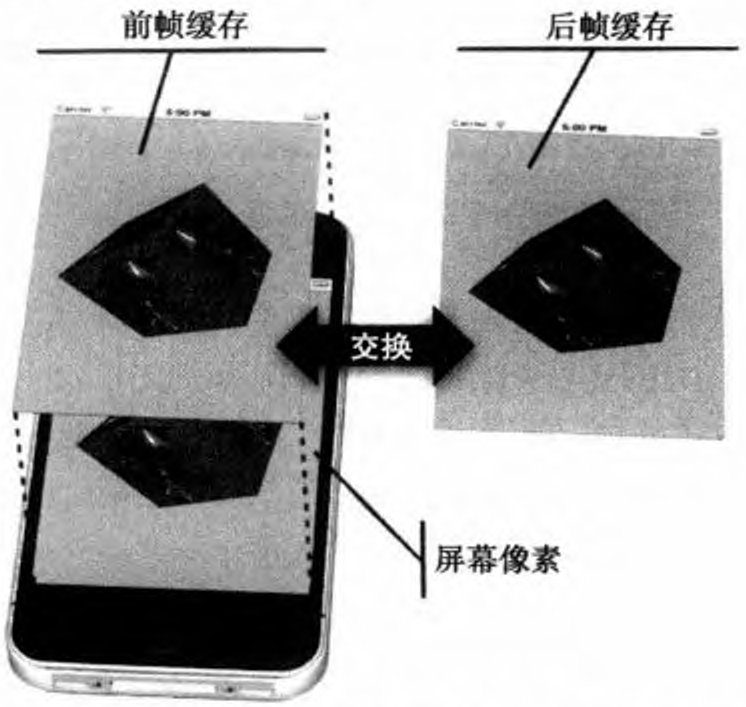


图 1-5　前帧缓存决定了屏幕上显示的像素颜色，同时会与后帧缓存切换

1.3　OpenGL ES 的上下文

用于配置 OpenGL ES 的保存在特定平台的软件数据结构中的信息会被封装到一个 OpenGL ES 上下文（context）中。OpenGL ES 是一个状态机器，这意味着在一个程序中设置了一个配置值后，这个值会一直保持，直到程序修改了这个值。上下文中的信息可能会被保存在 CPU 所控制的内存中，也可能会被保存在 GPU 所控制的内存中。OpenGL ES 会按需在两个内存区域之间复制信息，知道何时发生复制有助于程序的优化。第 9 章会介绍性能优化的技术。

OpenGL ES 上下文的内部实现依赖于特定的嵌入式系统以及特定的 GPU 硬件。OpenGL ES 为跟上下文的交互提供了 ANSI C 语言函数，以使程序不需要知道太多与特定系统相关的信息。

OpenGL ES 上下文会跟踪用于渲染的帧缓存。上下文还会跟踪用于几何数据、颜色等的缓存。上下文会决定是否使用某些功能，比如纹理和灯光，分别在第 3 章和第 4 章中进行讲解。上下文还会为渲染定义当前的坐标系统，这个会在第 2 章中讲解。

1.4　一个 3D 场景的几何数据

在为 GPU 提供数据时，很多种数据都可以省略，比如灯光信息和颜色。在渲染一个场景时，OpenGL ES 必须具有的一种数据是用来指定要渲染的图形的几何数据。几何数据是相对于 3D 坐标系定义的。

1.4.1　坐标系

图 1-6 展示的是 OpenGL 的坐标系。坐标系是用于帮助显示空间中的位置之间的关系的参考线的集合。图 1-6 中的每一个箭头叫做一个轴。OpenGL ES 总是开始于一个矩形的笛卡儿坐标系，这意味着任何两个轴之间的角度都是 90 度。空间中的每一个位置被称为一个顶点，每个顶点通过其在 X、Y、Z 轴上的位置被定义。

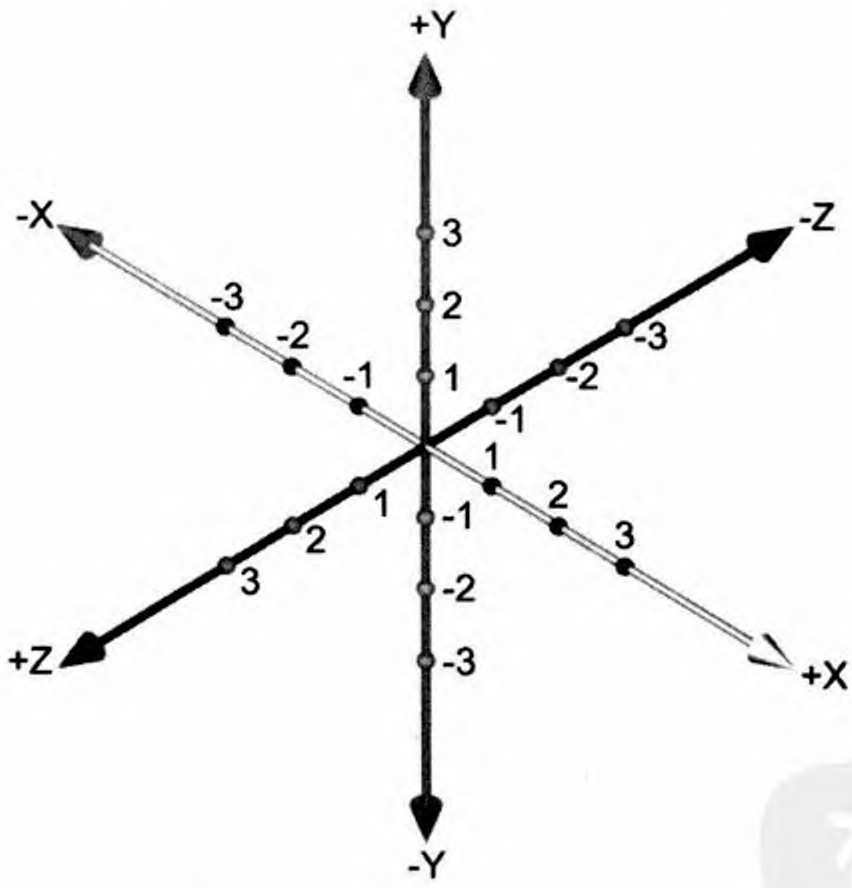


图 1-6　X、Y 和 Z 轴定义了 OpenGL 的坐标系

图 1-7 显示了相对于坐标轴的在 {1.5，3.0，0.0} 位置的顶点。这个顶点是通过 X 轴 1.5 的位置，Y 轴 3.0 的位置，Z 轴 0 的位置来定义的。图 1-7 中的虚线显示了顶点是怎么与坐标轴对齐的。

沿着每个轴的位置被称为坐标，确定一个用于 3D 图形的顶点需要 3 个坐标。图 1-8 展示了更多的顶点以及它们在 OpenGL 坐标系中的相对位置。虚线显示了顶点是怎么与坐标轴对齐的。

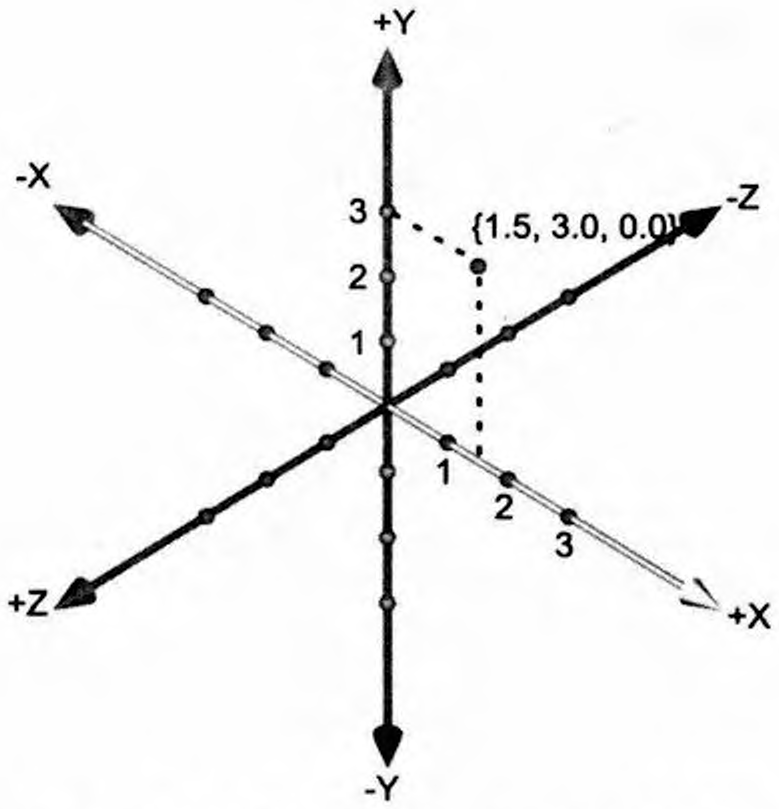


图 1-7　相对于坐标轴的在 {1.5，3.0，0.0} 位置的顶点

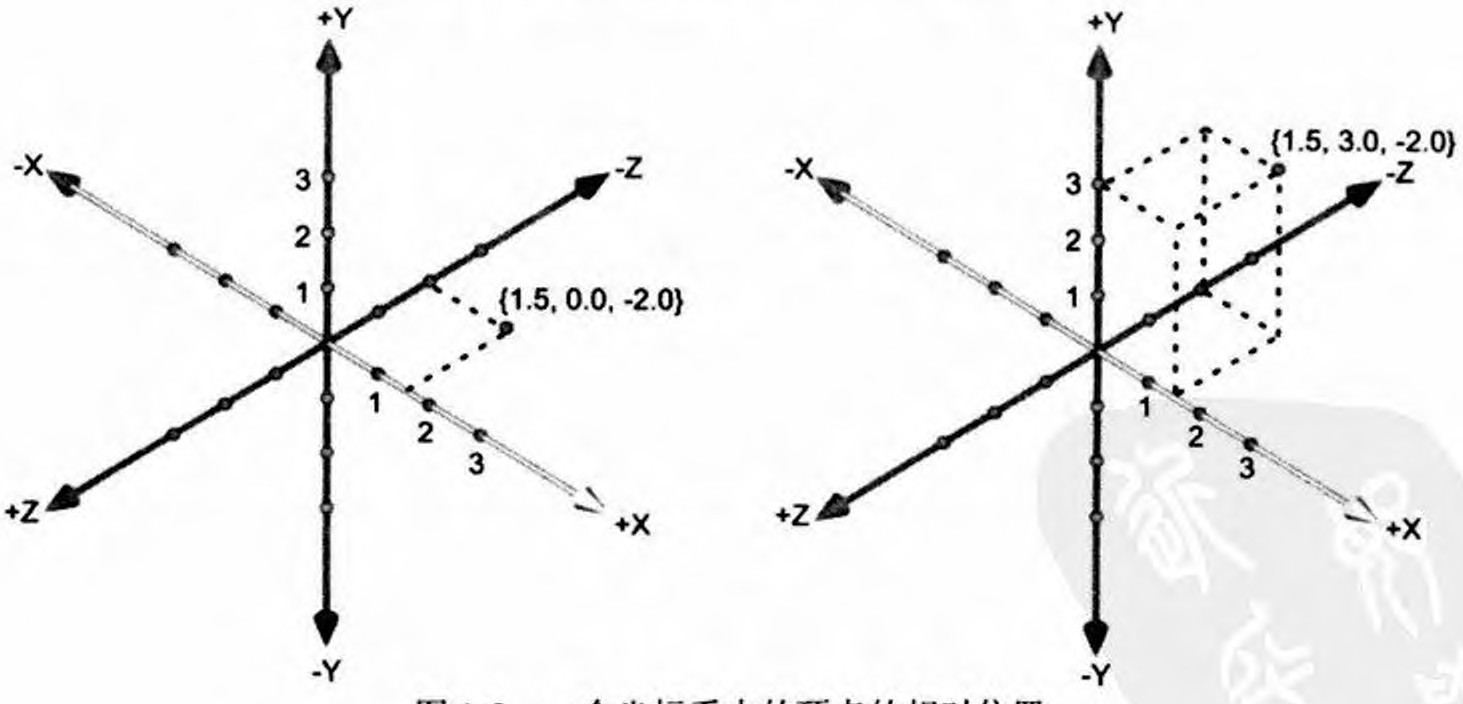


图 1-8　一个坐标系中的顶点的相对位置

OpenGL ES 坐标是以浮点数来存储的。现代 GPU 对浮点运算做了专门的优化，即使是使用其他数据类型的顶点也会被转换成浮点值。

使用和理解坐标系的关键点之一是记住它仅仅是一个假想的数学工具。第 5 章会介绍修改坐标系后的巨大影响。从纯数学意义上讲，可以有很多种非笛卡儿坐标系。例如，极坐标系是通过假想的落在一个使用两个角和一个半径的球的表面上的点来确定 3D 空间中的位置的。现在不用担心非笛卡儿坐标系的数学运算。硬件中的嵌入式 GPU

对于大部分非笛卡儿坐标系都是不支持的，并且本书中没有例子会使用非笛卡儿坐标系。如果你需要在工程中使用，以任何坐标系表达的位置都可以在需要的时候转换成 OpenGL ES 支持的默认坐标系。

OpenGL ES 坐标系没有单位。点 {1，0，0} 和点 {2，0，0} 之间的距离是沿着 X 轴的 1 单位，但是问下自己，“1 什么？是 1 英寸、1 毫米、1 英里，还是 1 光年？”答案是“这无关紧要”，确切地说这取决于你。你可以自由地假设距离 1 代表厘米还是你的 3D 场景中的其他单位。

注意　对于 3D 图形来说不定义单位可能非常方便，但是这也出现了一个挑战，就是在你想要打印你的 3D 场景的时候。苹果公司的 iOS 系统支持用于 2 维绘图兼容 PDF 的 Quartz 2D，并且按照现实世界的尺寸定义了单位。现实世界的尺寸允许你在绘制几何对象的同时知道将要绘制到与打印分辨率无关的页面上的对象的大小。与此相反，与分辨率无关的定义或者渲染 OpenGL 几何图形的方式是不存在的。

1.4.2　矢量

矢量是图形编程中频繁使用的另一个数学概念。从某种意义上讲，矢量是另一种诠释顶点数据的方式。矢量是既有方向又有距离的一个量。距离也叫大小。所有的顶点都可以用它相对于 OpenGL ES 坐标系原点（{0，0，0}）的距离和方向来定义。图 1-9 使用一个从原点到顶点 {1.5，3.0，–2.0} 的实心箭头来描述矢量。虚线显示了顶点是如何与坐标轴对齐的。

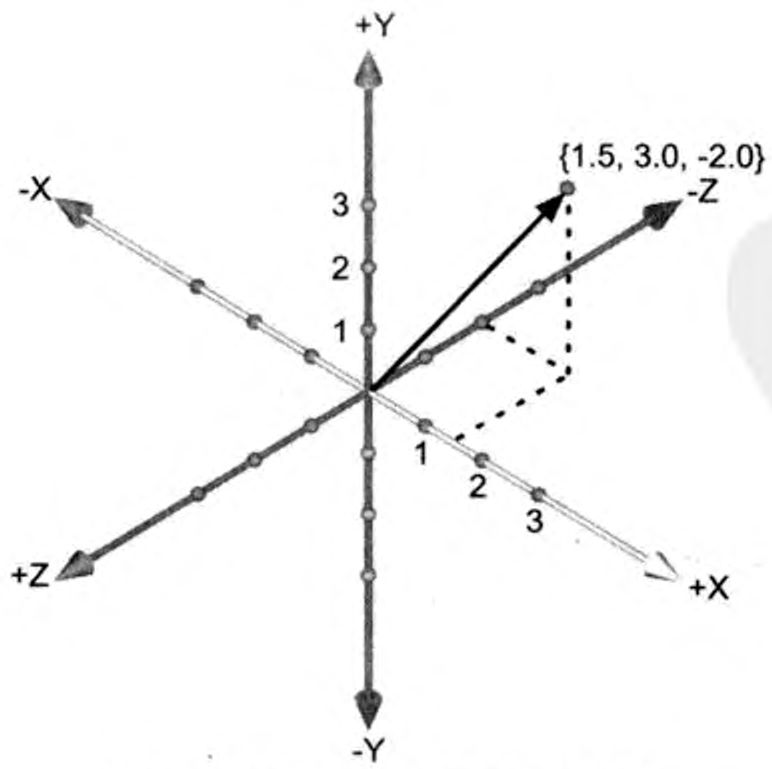


图 1-9　在 3D 坐标系中的一个矢量

可以使用每个顶点的坐标之间的差异来计算任意两个顶点之间的矢量。介于顶点{1.5，3.0，–2.0} 与原点之间的矢量是 {1.5–0.0，3.0–0.0，–2.0–0.0}。图 1-10 中的顶点 V1 与顶点 V2 之间的矢量等于 {V2.x – V1.x, V2.y – V1.y, V2.z – V1.z}。

矢量可以加在一起进而产生一个新矢量。介于原点与任意顶点之间的矢量是 3 个轴对齐矢量的和，参见图 1-11。矢量 A+B+C 等于矢量 D（如下式所示），同时确定了在{1.5，3.0，–2.0} 位置的顶点。

```
D.x = A.x + B.x + C.x = 1.5 + 0.0 + 0.0 = 1.5
D.y = A.y + B.y + C.y = 0.0 + 3.0 + 0.0 = 3.0
D.z = A.z + B.z + C.z = 0.0 + 0.0 + -2.0 = -2.0
```

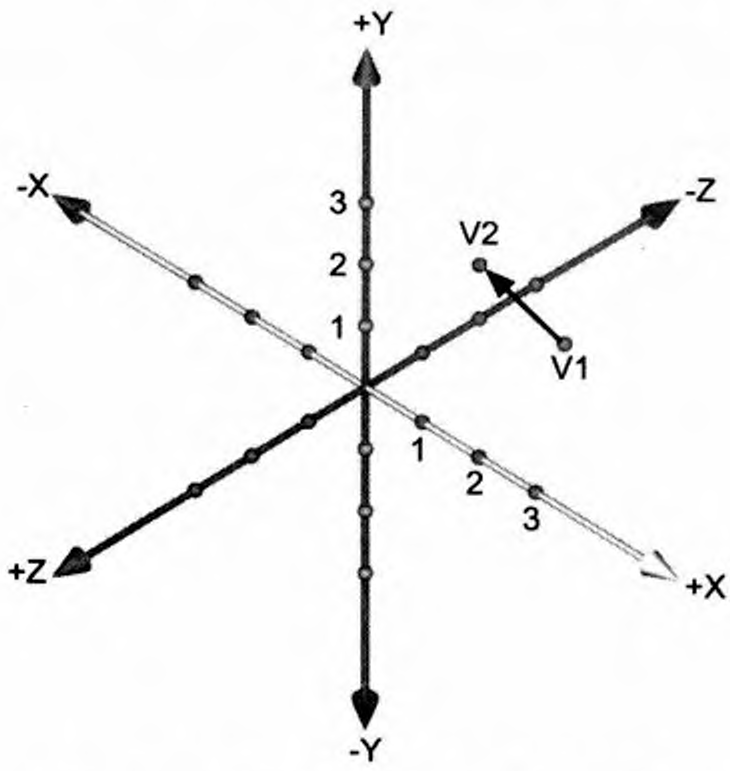


图 1-10　介于顶点 V1 与 V2 之间的矢量

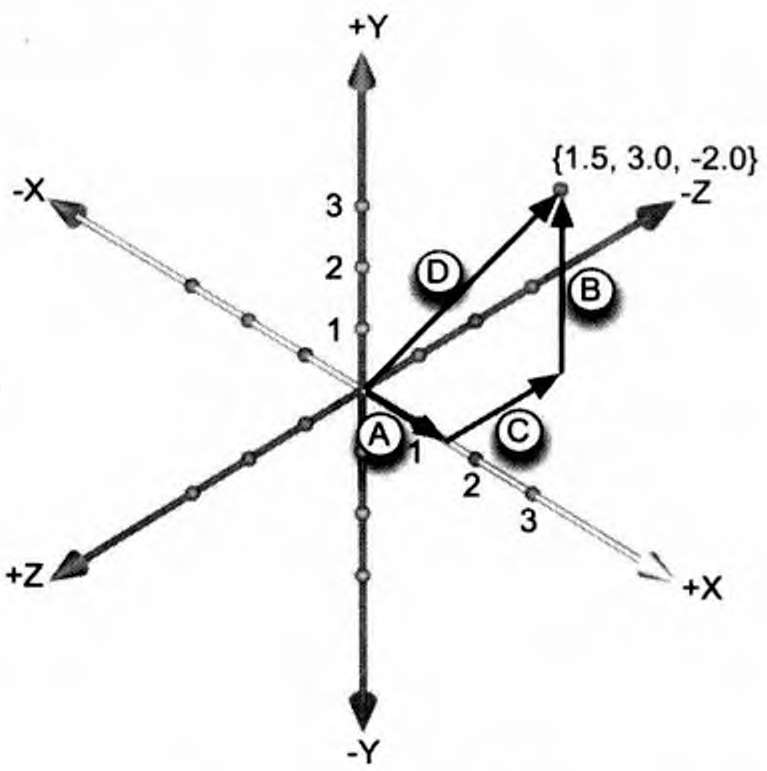


图 1-11　轴对齐矢量的和

矢量是理解现代 GPU 的关键，因为图形处理器就是大规模并行矢量处理器。GPU 能够同时控制多个矢量，并执行用于定义渲染结果的矢量运算。在后面的章节会根据需要适时地解释多个关键的除了加法和减法之外的矢量运算。OpenGL ES 的默认坐标系、顶点和矢量为要渲染的几何数据的定义提供了足够的数学元素。

注意　线性代数的整个领域是处理矢量数学运算。线性代数是关于三角学的，但是它主要使用简单的运算，如加法和乘法来建立和操作复杂的几何图形。计算机图形依赖于线性代数，因为计算机尤其是 GPU 擅长简单的数学运算。本书会根据需要逐渐对线性代数的概念进行讲解。

1.4.3　点、线、三角形

OpenGL ES 使用顶点数据来定义点、线段和三角形。一个顶点会定义坐标系中的一个点的位置，两个顶点会定义一个线段，三个顶点会定义一个三角形。OpenGL ES 只渲染顶点、线段和三角形。图 1-12 显示了使用很多三角形建立的几何对象有多复杂。

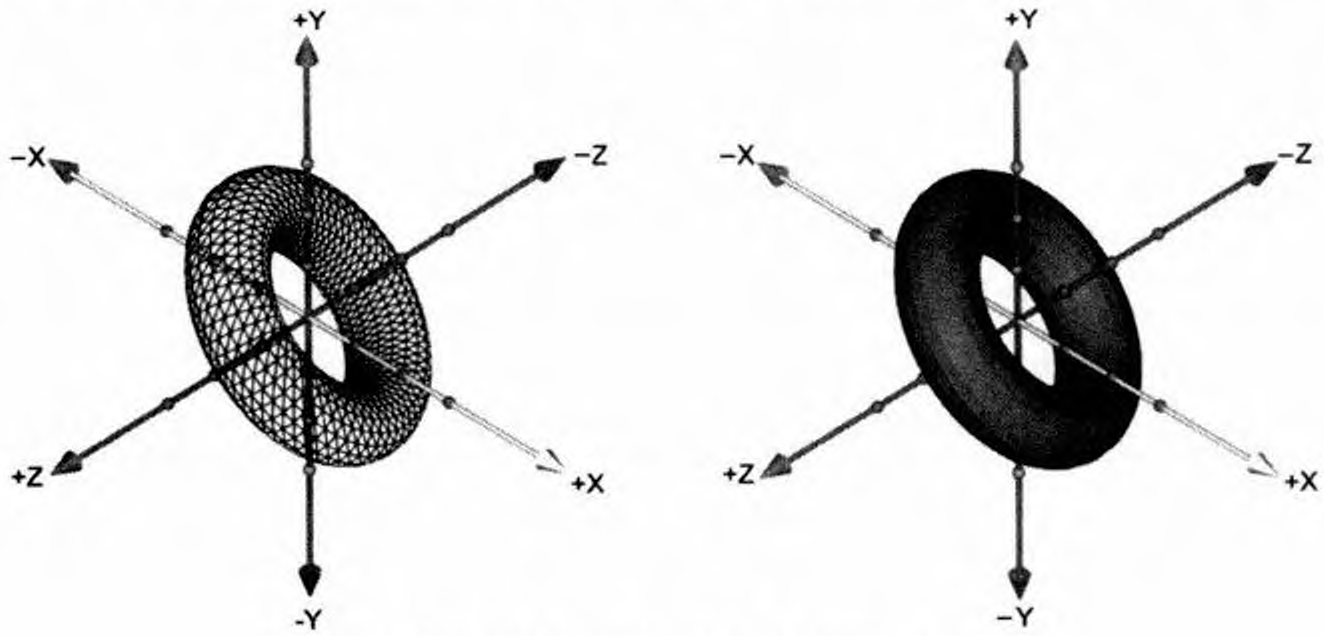


图 1-12　渲染为三角形和线段的顶点数据

1.5　小结

OpenGL ES 是访问类似 iPhone 和 iPad 的现代嵌入式系统的 3D 图形加速硬件的标准。把程序提供的几何数据转换为屏幕上的图像的过程叫做渲染。GPU 控制的缓存是高效渲染的关键。容纳几何数据的缓存定义了要渲染的点、线段和三角形。OpenGL ES 3D 的默认坐标系、顶点和矢量为几何数据的描述提供了数学基础。渲染的结果通常保存在帧缓存中。有两个特别的帧缓存，前帧缓存和后帧缓存，它们控制着屏幕像素的最终颜色。OpenGL ES 的上下文保存了 OpenGL ES 的状态信息，包括用于提供渲染数据的缓存地址和用于接收渲染结果的缓存地址。

第 2 章会介绍一个使用苹果 Xcode 开发工具和 Cocoa Touch 面向对象的框架在 iPhone、iPod Touch 和 iPad 中绘制 3D 图形的程序。第 2 章的例子是本书后面例子的基础。

第 2 章　让硬件为你工作

本章会讲解怎么在 iOS 5 应用中建立和使用 OpenGL ES 图形。一个初始例子程序会利用第 1 章中的图形概念让嵌入式硬件渲染一个图像。初始例子会被延伸成另外两个版本以解释苹果的 iOS 5 引入的 GLKit 技术与基础的 OpenGL ES 函数之间的关系。

2.1　使用 OpenGL ES 绘制一个 Core Animation 层

第 1 章介绍了 OpenGL ES 的帧缓存。iOS 操作系统不会让应用直接向前帧缓存或者后帧缓存绘图，也不会让应用直接控制前帧缓存和后帧缓存之间的切换。操作系统为自己保留了这些操作，以便它可以随时使用 Core Animation 合成器来控制显示的最终外观。

Core Animation 包含层的概念。同一时刻可以有任意数量的层。Core Animation 合成器会联合这些层并在后帧缓存中产生最终的像素颜色，然后切换缓存。图 2-1 显示的是合并两个层来产生后帧缓存中的颜色数据的过程。

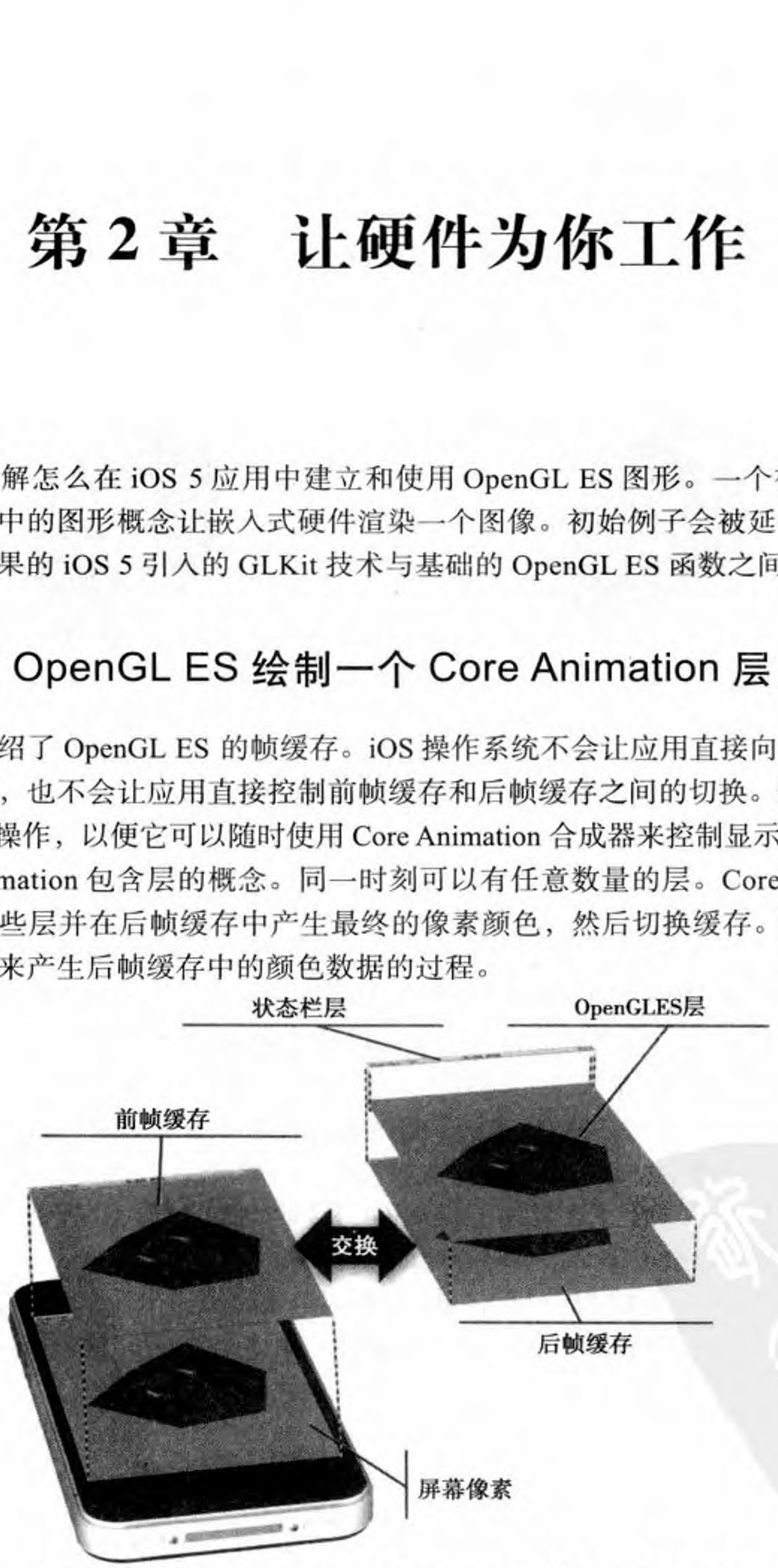


图 2-1　合并 Core Animation 层来产生后帧缓存中的颜色数据

一个应用提供的层与操作系统提供的层混合起来可以产生最终的显示外观。例如，在图 2-1 中，OpenGL ES 层显示了一个应用生成的旋转的立方体，但是在显示器顶部的显示状态栏的层是由操作系统生成和控制的。大部分的应用使用多个层。每一个 iOS 原生用户界面对象都有一个对应的 Core Animation 层，因此一个显示多个按钮、文本域、图像等的应用会自动使用多个层。

层会保存所有绘制操作的结果。例如，iOS 为有效地在层上绘制视频提供了软件对象。有用类似淡入淡出的特殊效果显示图像的层。层内容可以使用苹果的用于 2D 的支持富文本的 Core Graphics 框架来绘制。类似本章中的例子，应用会使用 OpenGL ES 来渲染层内容。

注意　苹果的 Core Animation 合成器使用 OpenGL ES 来尽可能高效地控制 GPU、混合层和切换帧缓存。图形程序员经常使用术语混合（composite）来描述混合图像以形成一个合成结果的过程。所有显示的图画都是通过 Core Animation 合成器来完成的，因此最终都涉及 OpenGL ES。

帧缓存会保存 OpenGL ES 的渲染结果，因此为了渲染到一个 Core Animation 层上，程序需要一个连接到某个层的帧缓存。简言之，每个程序用足够的内存配置一个层来保存像素颜色数据，之后创建一个使用层的内存来保存渲染的图像的帧缓存。图 2-2 介绍了 OpenGL ES 的帧缓存与层之间的关系。

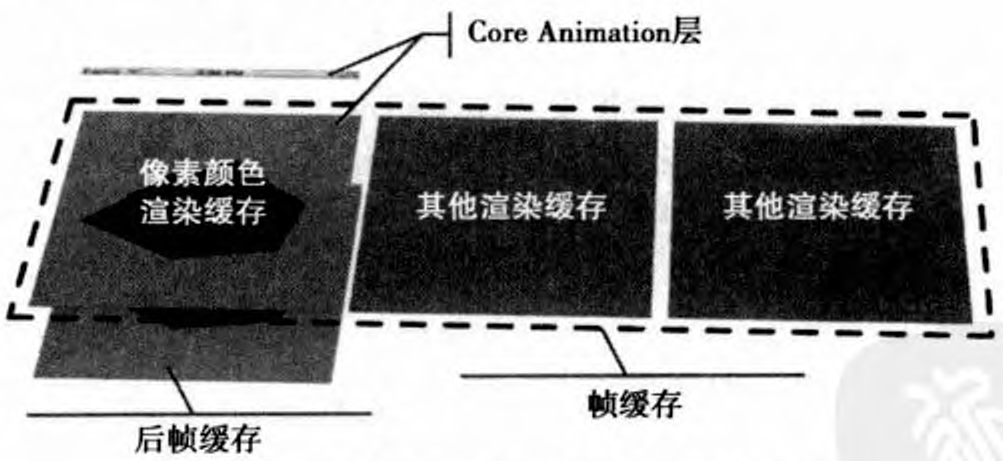


图 2-2　帧缓存可以与层分享像素存储器

图 2-2 显示了一个像素颜色渲染缓存（pixel color render buffer）和另外两个标识为其他渲染缓存（other render buffer）的缓存。除了像素颜色数据，OpenGL ES 和 GPU 有时会以渲染的副产品的形式产生一些有用的数据。帧缓存可以配置多个叫做渲染缓存的缓存来接收多种类型的输出。与层分享数据的帧缓存必须要有一个像素颜色渲染缓存。其他的渲染缓存是可选的，但是本章中不会使用。图 2-2 为完整起见显示了其他的渲染缓存，因为大部分不平凡的 OpenGL ES 程序会使用至少一个额外的渲染缓存，具体内容会在第 5 章中解释。

2.2 结合 Cocoa Touch 和 OpenGL ES

OpenGLES_Ch2_1 是本章的第一个示例应用，为本书的示例提供了起始点。这个程序设置 OpenGL ES 渲染一个图像到一个 Core Animation 层。之后由 iOS Core Animation 合成器自动地把渲染的层内容与其他层结合起来，产生保存在后帧缓存中的像素颜色数据，并最终显示到屏幕上。

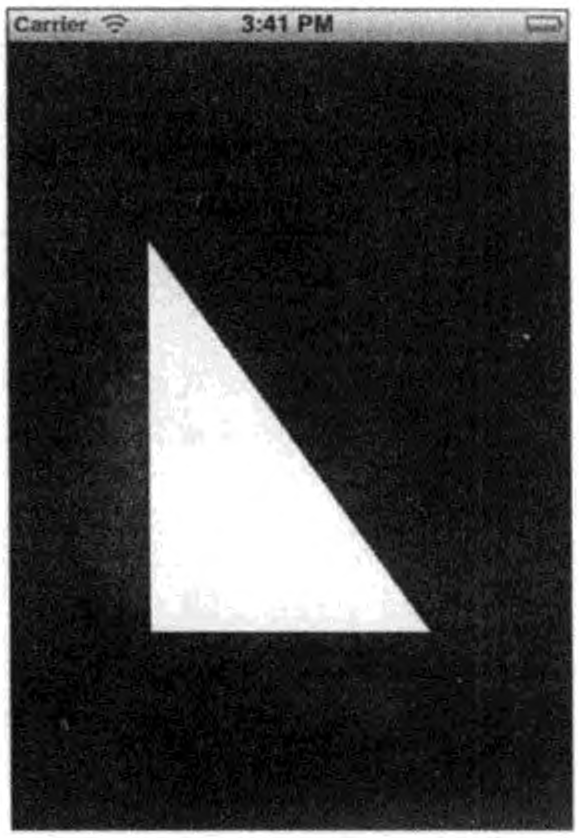


图 2-3 OpenGLES_Ch2_1 示例所产生的最终屏幕结果

图 2-3 显示了 OpenGLES_Ch2_1 渲染出来的图像。虽然只有一个三角形，但是执行 OpenGL ES 渲染的步骤与更复杂场景的步骤是相同的。这个例子使用了苹果的 Cocoa Touch 技术以及 Xcode 集成开发环境（IDE）。包含在 Xcode 中的苹果 iOS 开发者工具是 iOS 软件开发工具包（SDK）的一部分，其官方下载网址为 http://developer.apple.com/technologies/ios/。

2.2.1 Cocoa Touch

Cocoa Touch 由可重用的用于创建和运行应用的函数及软件对象组成。苹果的 iOS 由一个类似于 Mac OS X 的近乎完全与 UNIX 相似的操作系统组成，Mac OS X 是运行在苹果的 Macintosh 系列电脑上的一个操作系统。谷歌的基于 Linux 的 Android 操作系统也与 UNIX 类似。Cocoa Touch 建立在 UNIX 系统的基础之上，同时整合了许多不同的功能，从网络连接到 Core Animation，再到用户用来开始、停止、观察、与应用交互的图形用户界面。不使用 Cocoa Touch 而只使用 ANSI C 编程语言、UNIX 命令行工具，以及 UNIX 应用程序编程接口编写一个 iOS 程序在技术上是可能的。但是，大部分用户不会这么写程序，因为苹果不接受这样的应用发布到苹果的 App Store 上。

Cocoa Touch 主要是由 Objective-C 编程语言实现的。Objective-C 向 ANSI C 编程语言添加了少量的语法元素和一个面向对象的运行时系统。Cocoa Touch 提供了对于包括 OpenGL ES 的底层 ANSI C 技术的入口，但即使是最简单的像 OpenGLES_Ch2_1 这样的例子应用也需要 Objective-C。Cocoa Touch 提供了很多用于 iOS 应用的标准功能，以及让开发者专注于让应用变得独一无二的功能。每一个 iOS 程序员都受益于对 Cocoa Touch 和 Objective-C 的学习和使用。但是，本书主要关注 OpenGL ES，所以仅仅简要介绍 Cocoa Touch 技术。

2.2.2　使用苹果开发者工具

Xcode 运行在 Mac OS X 上，包含可识别语法的代码编辑器、编译器、调试器、性能工具，以及一个文件管理用户界面。Xcode 支持使用 ANSI C、C++、Objective-C 和 Objective-C++ 语言做开发，同时它可以与各种外部源代码管理系统一起工作。苹果使用 Xcode 开发自己的软件。更多信息可登录：http://developer.apple.com/iphone/library/referencelibrary/GettingStarted/URL_Tools_for_iPhone_OS_Development/index.html 查询。

图 2-4 显示的是 Xcode，并且加载了用 OpenGLE_Ch2_1.xcodeproj 配置文件定义的建立 OpenGLES_Ch2_1 例子所需要的资源。Xcode 有与大部分其他的 IDE 相似的功能，例如开源 Eclipse IDE 和微软的 Visual Studio IDE。图 2-4 中左边栏的列表指定了需要编译并链接到最终应用中的文件。在 Xcode 窗口顶部的工具栏提供了用于组建和运行开发中的应用的按钮。剩下的用户界面主要是由源代码编辑器组成。

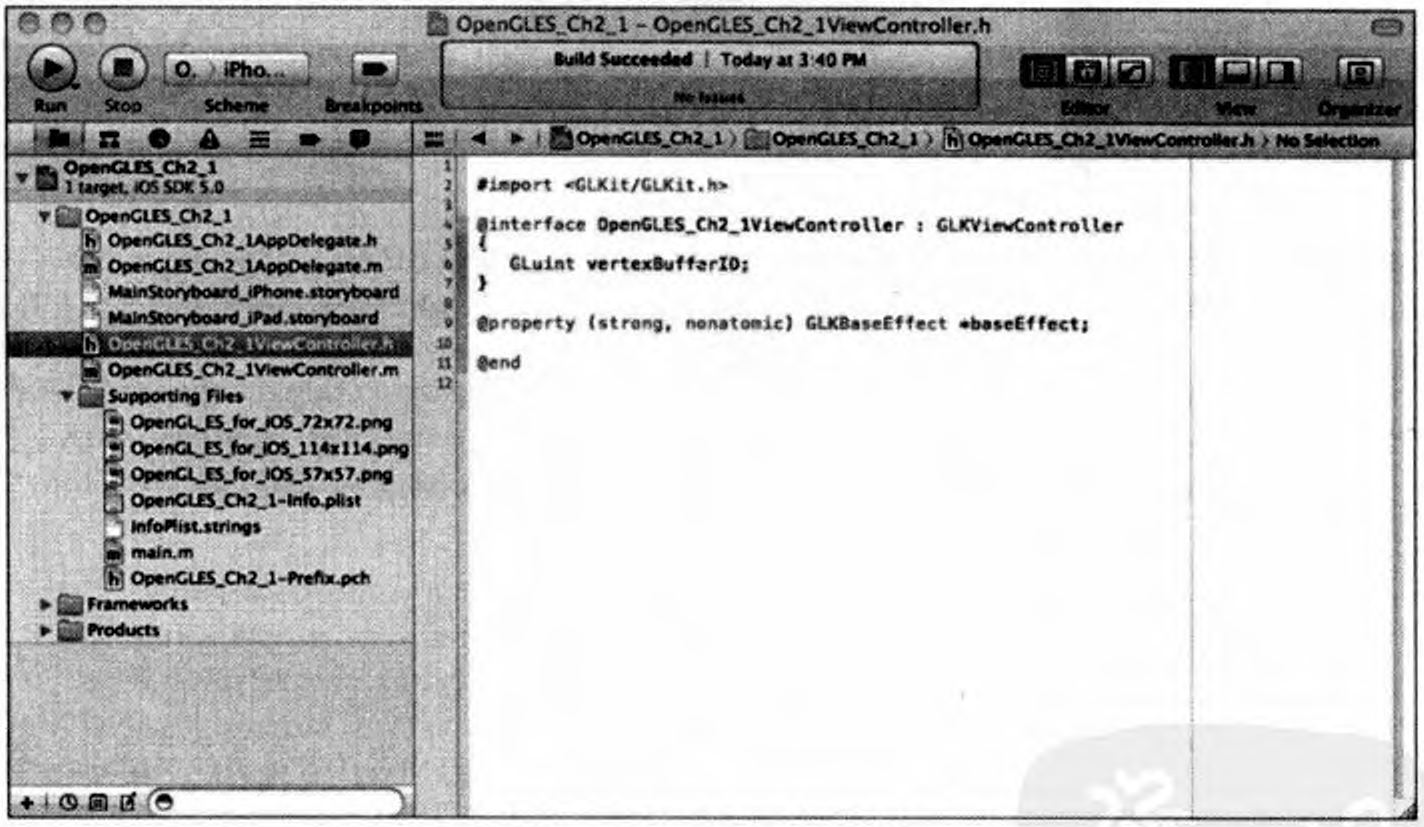


图 2-4　Xcode 工程示例

2.2.3　Cocoa Touch 应用架构

图 2-5 列出了所有使用 OpenGL ES 的最新 Cocoa Touch 应用的主要软件组件。箭头指示了组件之间的典型的信息流。在图 2-5 中 Cocoa Touch 提供了灰色的组件，应用通常使用不需修改的灰色组件。图 2-5 中的白色组件对于每一个应用来说都是独一无二的。较复杂的应用会包含专属的软件组件。不要被图 2-5 的复杂所迷惑。大部分情况下，只有应用委托（application delegate）和根视图控制器（root view controller）这两个白色组件需要所有程序员的干预，其他组件是提供标准的 iOS 应用行为的基础结构的

一部分，均不需要程序员做任何干预。

操作系统控制对硬件组件的访问，发送用户的屏幕触摸事件给基于 Cocoa Touch 的应用。Cocoa Touch 实现了标准的图形组件，包括触摸键盘和状态栏，因此单独的应用不需要重新创建这些组件。苹果提供了一个图表以说明常用的 Cocoa Touch 应用组件，图表地址为：http://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/AppArchitecture/AppArchitecture.html。为简单起见，苹果的图表省略了 OpenGL ES 和 Core Animation 层，但是却为 Cocoa Touch 应用设计提供了额外的依据。

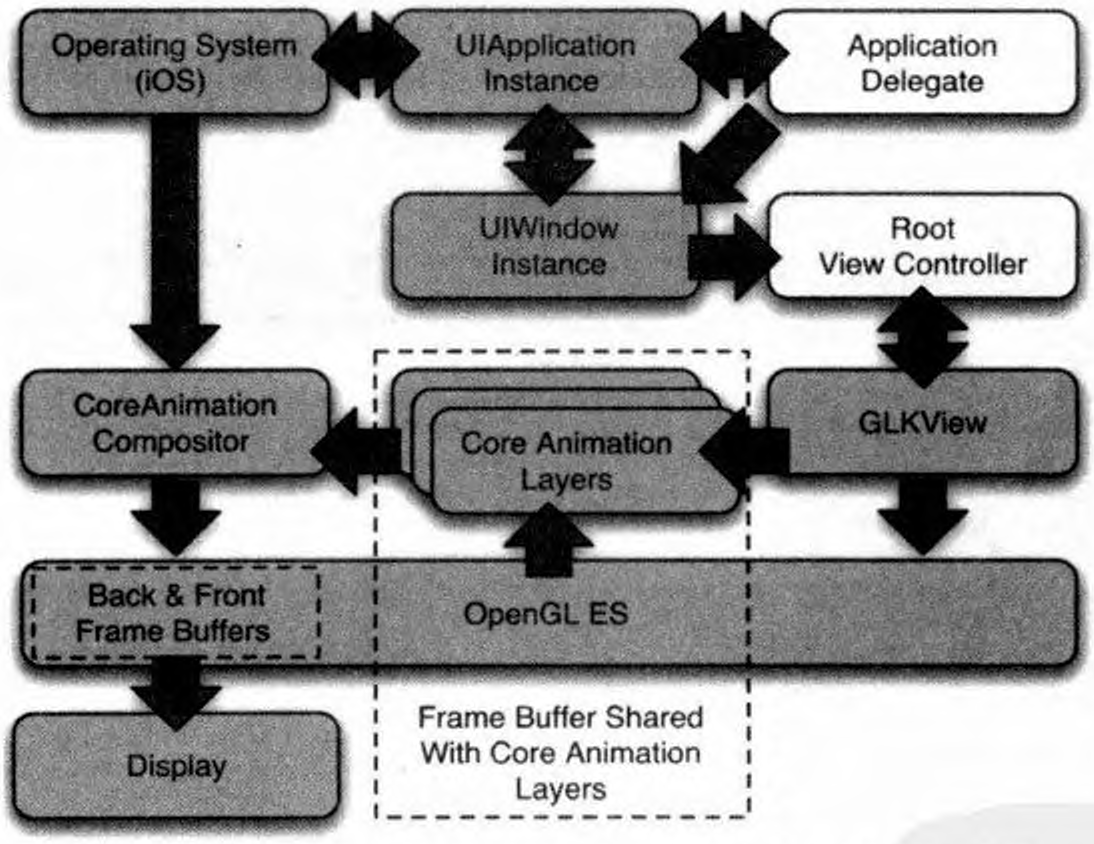


图 2-5　Cocoa Touch OpenGL ES 应用的软件架构

图 2-5 中显示了第 1 章介绍的 OpenGL ES 和帧缓存组件的作用。本章介绍了 Core Animation 层和 Core Animation 合成器。图 2-5 中剩下的组件实现了标准的 Cocoa Touch 行为。

- UIApplication：每个应用都包含一个单一的 UIApplication 类的实例。UIApplication 是一个 Objective-C Cocoa Touch 对象，它提供了应用与 iOS 之间的双向通信。应用向 iOS 请求服务，系统为正在运行的应用提供信息，例如显示器的当前方向。UIApplication 会与一个或更多个 Cocoa Touch UIWindow 实例通信，还会与一个用于路由用户输入事件到正确的对象的委托（delegate）通信。
- 应用委托 (application delegate)：委托对象提供了一个响应另一个对象的变化或者影响另一个对象的行为的机会。基本思路是两个对象协调解决一个问题。对象

（如 UIApplication）是非常常见并可以在各种情况下重用的。它保存了一个对于另一个对象（它的委托）的引用，并且在关键时刻发送消息给委托。可能只是通知委托发生了一些事情并给予委托一个执行额外处理的机会的消息，或者是要求委托返回能够控制发生什么的关键信息的消息。委托通常是一个特定的应用所特有的自定义对象。应用委托会接收关于 Cocoa Touch 应用所运行的环境的所有重要改变，包括应用完成启动和结束这样的改变。

- UIWindow：Cocoa Touch 应用总是有至少一个自动创建的覆盖整个屏幕的 UIWindow 实例。UIWindow 实例控制屏幕的矩形区域，并且它们能够被重叠和分层以便一个窗口覆盖另一个窗口。除了覆盖整个屏幕的窗口，Cocoa Touch 应用很少直接访问窗口。Cocoa Touch 会按需自动使用其他的 UIWindows 来向用户显示警告和状态信息。UIWindows 会包含一个或多个提供窗口图形内容的 UIView 实例。应用架构内的窗口的一个重要的作用是从 UIApplication 实例收集用户的输入事件，然后根据实际情况把这些事件重新发送给正确的 UIView 实例。例如，UIWindow 确定哪一个 UIView 实例被用户触摸，然后直接发送合适的事件到那个实例。
- 根视图控制器：每一个窗口都有一个可选的根视图控制器。视图控制器是 Cocoa Touch UIViewController 类的实例并把大部分 iOS 应用的设计联系起来。视图控制器会调节一个相关的视图的外观并支持在设备方向变化时旋转视图。根视图控制器指定填充整个窗口的 UIView 实例。UIViewController 类的默认行为控制了 iOS 应用的标准可视化效果。GLKViewController 类是支持 OpenGL ES 特有的行为和动画计时的 UIViewController 的内建子类。OpenGLES_Ch2_1 例子创建了一个 GLKViewController 的子类 OpenGLES_Ch2_1ViewController 来提供所有例子所特有的行为。
- GLKView：这是 Cocoa Touch UIView 类的内建子类。GLKView 简化了通过用 Core Animation 层来自动创建并管理帧缓存和渲染缓存共享内存所需要做的工作。GLKView 相关的 GLKViewController 实例是视图的委托并接收当视图需要重绘时的消息。创建你自己的 UIView 或者 GLKView 的子类来实现应用特有的绘图是可能的，但是 OpenGLES_Ch2_1 采用了简单的方法，使用未经修改的 GLKView。OpenGLES_Ch2_1 例子实现了所有应用特有的行为，包括在 OpenGLES_Ch2_1ViewController 类中的绘画。

注意　GLKView 和 GLKViewController 类名字中的 GLK 前缀表明这些类是 iOS 5 引入的 GLKit 框架的一部分。框架是指被编译后的代码所使用的编译后的代码、接口声明，以及类似图像和数据的资源文件的集合。框架高效地组织可重用共享库并且在一些情况下可能包含库的多个版本。GLKit 为简化 iOS 中 OpenGL ES 的使用提供了类和函数。GLKit 是 Cocoa Touch 以及多个其他的框架（包含 UIKit）的一部分，UIKit 包含许多类如 UIApplication、UIWindow 和 UIView。

2.3 OpenGLES_Ch2_1 示例

你可以从网址 http://opengles.cosmicthump.com/learning-opengl-es-sample-code/ 上下载关于本书的示例代码。Xcode 工程以及所有创建本书的示例所需要的文件都包含在里面。一个叫做 OpenGLES_Ch2_1.xcodeproj 的文件保存了关于工程自身的信息。在电脑上正确地安装了苹果的 iOS 5 软件开发工具包和 Xcode 之后，双击 OpenGLES_Ch2_1.xcodeproj 文件来开启 Xcode 并加载工程。在加载完成后，点击 Xcode 工具栏的 Run 按钮来编译并连接工程中的文件，然后会启动苹果的 iPhone 模拟器来运行 OpenGLES_Ch2_1 应用。

注意 本书的所有例子都是用苹果的 ARC 技术来为 Objective-C 对象管理内存的。在新创建的 iOS Xcode 工程中会默认开启 ARC。使用 ARC 避免了手动为对象管理内存并简化了例子代码。

图 2-6 列出了例子中构建和连接的文件。本节的其余部分介绍了每个文件的内容和目的。

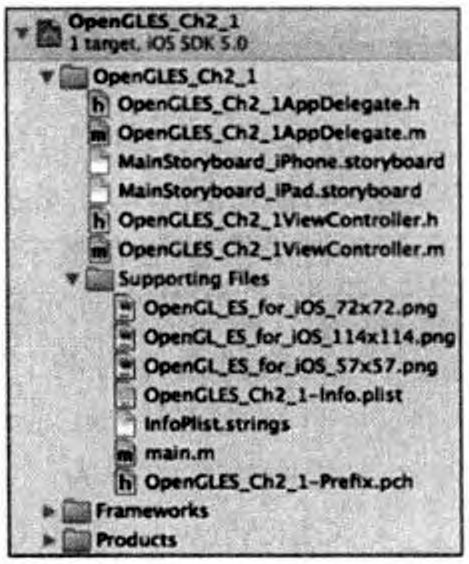


图 2-6 在 OpenGLES_Ch2_1 Xcode 工程中的文件

OpenGLES_Ch2_1 工程是使用 Xcode 的标准单视图应用（Single View Application）模板创建的。这个模板设置新工程以创建一个由一个单独的填满整个屏幕的 UIView（或者其子类）实例组成的简单应用。其他模板为其他类型的 iOS 应用提供了一个起始点。单视图应用模板生成了一个命名为 OpenGLES_Ch2_1AppDelegate 的自定义应用委托类，一个命名为 OpenGLES_Ch2_1ViewController 的自定义视图控制器类。

2.3.1 OpenGLES_Ch2_1AppDelegate 类

OpenGLES_Ch2_1AppDelegate.h 文件是在工程刚创建时由 Xcode 自动生成的。OpenGLES_Ch2_1AppDelegate.h 文件包含对 OpenGLES_Ch2_1AppDelegate 类的 Objective-C 声明。OpenGLES_Ch2_1 例子可以不用做任何修改使用生成的类。

OpenGLES_Ch2_1AppDelegate.m 文件也是在工程刚创建时由 Xcode 自动生成的。OpenGLES_Ch2_1AppDelegate.m 文件包含对 OpenGLES_Ch2_1AppDelegate 类的 Objective-C 实现代码。生成的代码包含多个一般由应用的委托实现的方法的存根实现。OpenGLES_Ch2_1 例子可以不用做任何修改使用生成的类。

2.3.2　Storyboards

MainStoryboard_iPhone.storyboard 和 MainStoryboard_iPad.storyboard 文件也是由 Xcode 的单视图应用模板生成的。在 Xcode 中可以可视化地编辑这些文件以设定用户界面。iPad 与 iPhone 的屏幕大小不同，因此会有不同的用户界面。当例子运行时，Open-GLES_Ch2_1 示例会自动读取与当前设备相适合的 Storyboard。Storyboards 会把 UIViewController 实例以及与其相关联的 UIView 实例联系起来。Storyboards 会指定视图控制器之间的过渡，自动化应用的大部分用户交互设计，潜在地消除了原本需要编写的代码。但是，OpenGLES_Ch2_1 这个例子是如此简单以至于它只使用了一个视图控制器——OpenGLES_Ch2_1ViewController 类的一个实例。

2.3.3　OpenGLES_Ch2_1ViewController 类的 interface

OpenGLES_Ch2_1ViewController.h 文件原本是在工程刚创建的时候由 Xcode 生成的，但是在这个例子中做了一些修改。OpenGLES_Ch2_1ViewController.h 包含修改过的用于 OpenGLES_Ch2_1ViewController 类的 Objective-C 声明。粗体的代码是对所生成代码的主要修改。

```
//
//  OpenGLES_Ch2_1ViewController.h
//

#import <GLKit/GLKit.h>

@interface OpenGLES_Ch2_1ViewController : GLKViewController
{
   GLuint vertexBufferID;
}

@property (strong, nonatomic) GLKBaseEffect *baseEffect;

@end
```

文件以一段注释开始。Cocoa Touch GLKit 框架接口是通过 #import 编译指令导入的，这与 ANSI C 的 #include 指令类似。两个指令都把特定文件的内容插入到包含这些指令的编译文件中。当相同的文件内容被导入到每个编译文件的次数超过一次时，Objective-C 的 #import 指令就会自动将其阻止，这里首选 #import，虽然 Objective-C 也支持 #include。

OpenGLES_Ch2_1ViewController 是 GLKViewController 类的一个子类并从 GLKViewController 继承了很多基本功能，GLKViewController 又从它的超类 UIViewController 继承了很多

功能。特别地，GLKViewController 会自动地重新设置 OpenGL ES 和应用的 GLKView 实例以响应设备方向的变化并可视化过渡效果，例如淡出和淡入。

在 OpenGLES_Ch2_1ViewController 的接口中声明的 vertexBufferID 变量保存了用于盛放本例中用到的顶点数据的缓存的 OpenGl ES 标识符。OpenGLES_Ch2_1View-Controller 类的实现代码解释了缓存标识符的初始化和使用。

OpenGLES_Ch2_1ViewController 接口中的 baseEffect 属性声明了一个 GLKBaseEffect 实例的指针。Objective-C 属性声明了与实例变量相似的值。Objective-C 对象的属性可以使用“点符号”访问，例如，someObject.baseEffect；或者使用方法访问，比如按照 -set<PropertyName> 命名约定的用于设定属性值的方法和按照 -<propertyName> 命名约定的用于返回属性值的方法。这些特别命名的方法叫做访问器（accessor）。-baseEffect 访问器用于返回 OpenGLES_Ch2_1ViewController 的 baseEffect 属性的值。用于设置这个属性值的访问器是“-setBaseEffect:”。属性通常并不一定是以实例变量的形式实现的，它们的值可能是按需计算的或者是从数据库里加载的。Objective-C 的属性语法提供了对象不需要暴露类声明中的值是怎么保存的就能提供值的声明方式。当点符号被用来获取或设置一个属性的值时，Objective-C 编译器会自动地替换成对于适当命名的访问器方法的访问。Objective-C 还提供了一个自动生成访问器方法的实现代码的方法，这个我们会在 OpenGLES_Ch2_1ViewController 的实现代码中讲到。

GLKBaseEffect 是 GLKit 提供的另一个内建类。GLKBaseEffect 的存在是为了简化 OpenGL ES 的很多常用操作。GLKBaseEffect 隐藏了 iOS 设备支持的多个 OpenGL ES 版本之间的差异。在应用中使用 GLKBaseEffect 能减少需要编写的代码量。在 OpenGLES_Ch2_1ViewController 的实现中详细解释了 GLKBaseEffect。

2.3.4 OpenGLES_Ch2_1ViewController 类的实现

OpenGLES_Ch2_1ViewController.m 文件是原先那个工程刚建立时 Xcode 自动生成的文件，不过为这个例子做了一些修改。OpenGLES_Ch2_1ViewController.m 文件包含了针对 OpenGLES_Ch2_1ViewController 类的 Objective-C 的实现。在实现代码中只定义了 3 个方法：-viewDidLoad、-glkView:drawInRect: 和 -viewDidUnload。本节会详细解释这 3 个方法。本章和后面章节中的例子会在这个例子的源码之上建立。以下就是实现代码的开始部分：

```
//
//  OpenGLES_Ch2_1ViewController.m
//  OpenGLES_Ch2_1
//

#import "OpenGLES_Ch2_1ViewController.h"
```

```
@implementation OpenGLES_Ch2_1ViewController

@synthesize baseEffect;
```

@synthesize baseEffect；表达式让 Objective-C 编译器为 baseEffect 属性自动地生成访问器方法。相对于使用 @synthesize 表达式，另一种方法是在代码中显式实现恰当命名的访问器方法。对于这个例子来说没有显式实现其访问器的任何理由，因为标准的访问器行为就足够了。只有在属性的存储需要特殊处理时或者属性值的变化需要调用自定义应用逻辑时才需要显式地编写访问器。

在 OpenGLES_Ch2_1ViewController.m 中的接下来一段代码定义了一个 C 结构体 SceneVertex，用来保存一个 GLKVector3 类型的成员 positionCoords。回顾一下第 1 章，顶点位置可以用一个起始于坐标系原点的矢量来表示。GLKit 的 GLKVector3 类型保存了 3 个坐标：X、Y 和 Z。

vertices 变量是一个用顶点数据初始化的普通 C 数组，这个变量用来定义一个三角形。

The `vertices` variable is as an ordinary C array initialized with vertex data to define a triangle.

```
//////////////////////////////////////////////////////////////////
// This data type is used to store information for each vertex
typedef struct {
   GLKVector3  positionCoords;
}
SceneVertex;

//////////////////////////////////////////////////////////////////
// Define vertex data for a triangle to use in example
static const SceneVertex vertices[] =
{
   {{-0.5f, -0.5f, 0.0}}, // lower left corner
   {{ 0.5f, -0.5f, 0.0}}, // lower right corner
   {{-0.5f,  0.5f, 0.0}}  // upper left corner
};
```

这个例子的顶点位置坐标是挑选出来的，因为默认的用于一个 OpenGL 上下文的可见坐标系是分别沿着 X、Y、Z 轴从 –1.0 延伸到 1.0 的。例子三角形的坐标把它置于可见坐标系的中央并且与 X 和 Y 轴所形成的平面对齐。图 2-7 显示了 vertices[] 定义的在一个代表默认 OpenGL ES 坐标系的可见部分的立方体内的三角形。

1. -viewDidLoad

接下来的 -viewDidLoad 方法为 OpenGL ES 提供三角形的顶点数据。-viewDidLoad 方法继承自 GLKViewController 类，当与 GLKViewController 相关联的应用的 GLKView 实例被从 storyboard 文件加载时会自动调用这个方法。OpenGLES_Ch2_1ViewController

提供了它自己的 -viewDidLoad 方法的实现，在这个实现中会首先调用它的超类的实现：

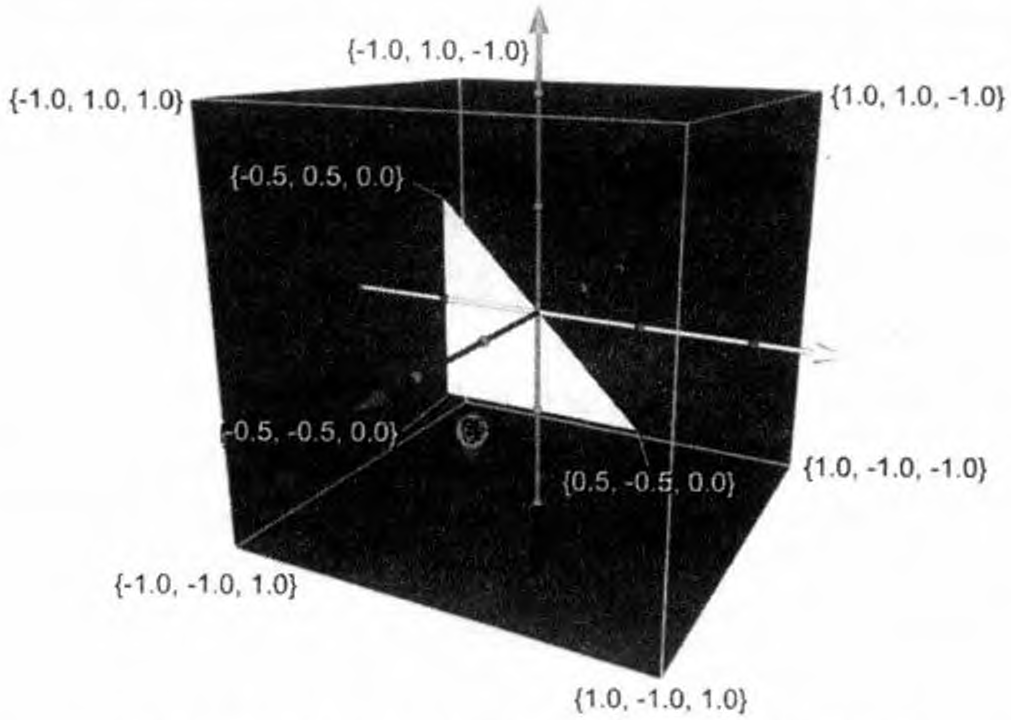


图 2-7　在默认的 OpenGL ES 坐标系中的三角形顶点

```
////////////////////////////////////////////////////////////////
// Called when the view controller's view is loaded
// Perform initialization before the view is asked to draw
- (void)viewDidLoad
{
   [super viewDidLoad];

// Verify the type of view created automatically by the
// Interface Builder storyboard
GLKView *view = (GLKView *)self.view;
NSAssert([view isKindOfClass:[GLKView class]],
   @"View controller's view is not a GLKView");

// Create an OpenGL ES 2.0 context and provide it to the
// view
view.context = [[EAGLContext alloc]
   initWithAPI:kEAGLRenderingAPIOpenGLES2];

// Make the new context current
[EAGLContext setCurrentContext:view.context];

// Create a base effect that provides standard OpenGL ES 2.0
// Shading Language programs and set constants to be used for
// all subsequent rendering
self.baseEffect = [[GLKBaseEffect alloc] init];
```

```
    self.baseEffect.useConstantColor = GL_TRUE;
    self.baseEffect.constantColor = GLKVector4Make(
       1.0f, // Red
       1.0f, // Green
       1.0f, // Blue
       1.0f);// Alpha

    // Set the background color stored in the current context
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // background color

    // Generate, bind, and initialize contents of a buffer to be
    // stored in GPU memory
    glGenBuffers(1,                 // STEP 1
       &vertexBufferID);
    glBindBuffer(GL_ARRAY_BUFFER,  // STEP 2
       vertexBufferID);
    glBufferData(                   // STEP 3
       GL_ARRAY_BUFFER,  // Initialize buffer contents
       sizeof(vertices), // Number of bytes to copy
       vertices,         // Address of bytes to copy
       GL_STATIC_DRAW);  // Hint: cache in GPU memory
}
```

-viewDidLoad 方法会将它继承的 view 属性的值转换为 GLKView 类型。类似 OpenGLES_Ch2_1ViewController 的 GLKViewController 的子类只能与 GLKView 实例或者是 GLKView 子类的实例一起正确工作。但是，这个例子的 storyboard 文件定义了哪一个是与应用的 GLKViewController 实例相关联的视图。使用 NSAssert() 函数的一个运行时检查会验证在运行时从 storyboard 加载的视图是否确实是正确的类型。如果验证的条件为 false，那么 NSAssert() 会向调试器或 iOS 设备控制台发送一个错误消息。NSAssert() 还会生成一个如果不做处理就停止应用的 NSInternalInconsistencyException。在这个例子中，无法从一个加载自 storyboard 的错误视图还原应用的界面，因此在运行时监测到错误的时候最好先停止应用。

如在第 1 章介绍的，OpenGL ES 的上下文不仅会保存 OpenGL ES 的状态，还会控制 GPU 去执行渲染运算。OpenGLES_Ch2_1ViewController 的 -viewDidLoad 方法会分配并初始化一个内建的 EAGLContext 类的实例，这个实例会封装一个特定于某个平台的 OpenGL ES 上下文。苹果还没有说明开头的 EAGL 前缀代表什么，但是它可能代表的是“Embedded Apple GL”。苹果 iOS 中的 OpenGLES 框架一般是以 EAGL 为前缀来声明 Objective-C 类和函数的。

在任何其他的 OpenGL ES 配置或者渲染发生之前，应用的 GLKView 实例的上下文属性都需要设置为当前。EAGLContext 实例既支持 OpenGL ES 1.1，又支持 OpenGL ES 2.0。本书中的例子使用的是 2.0 版本。下面的代码行在为视图的上下文属性赋值之

前，分配了一个新的 EAGLContext 的实例，并用粗体标注的常量将它初始化为 OpenGL ES 2.0。

```
view.context = [[EAGLContext alloc]
   initWithAPI:kEAGLRenderingAPIOpenGLES2];

// Make the new context current
[EAGLContext setCurrentContext:view.context];
```

一个应用可以使用多个上下文。EAGLContext 的方法“+setCurrentContext:”会为接下来的 OpenGL ES 运算设置将会用到的上下文。“+setCurrentContext:”方法前面的加号表明“+setCurrentContext:”是一个类方法。在 Objective-C 中，类方法是在即使没有这个类的实例时仍然可以被类自身调用的方法。

苹果的 OpenGLES.framework 为 EAGLContext 的“-initWithAPI:”方法声明了常量 kEAGLRenderingAPIOpenGLES2。同时，还存在着一个 kEAGLRendering-APIOpen-GLES1 常量。OpenGL ES 2.0 标准与以前的版本有很大的不同。尤其是 OpenGL ES 2.0 省略了很多特性和在上一个标准中定义的应用支持基础结构。作为替代，OpenGL ES 2.0 提供了一个新的更灵活的可编程 GPU 概念。为了灵活性，苹果建议我们在新应用中使用 OpenGL ES 2.0。在介绍苹果的 GLKit 之前，OpenGL ES 2.0 还需要一些前期工作以编程 GPU 并重新创建一些 2.0 版本丢失的、1.1 版本默认包含的便利功能。GLKit 现在替换了大部分 OpenGL ES 1.1 的基础结构，同时让 OpenGL ES 2.0 与 OpenGL ES 1.1 使用起来一样容易。

-viewDidLoad 方法接着设置 OpenGLES_Ch2_1ViewController 的 baseEffect 属性为一个新分配并初始化的 GLKBaseEffect 类型的实例，同时设置 GLKBaseEffect 实例的一些属性为比较适合这个例子的值。

```
// Create a base effect that provides standard OpenGL ES 2.0
// Shading Language programs and set constants to be used for
// all subsequent rendering
self.baseEffect = [[GLKBaseEffect alloc] init];
self.baseEffect.useConstantColor = GL_TRUE;
```

GLKBaseEffect 类提供了不依赖于所使用的 OpenGL ES 版本的控制 OpenGL ES 渲染的方法。OpenGL ES 1.1 跟 OpenGL ES 2.0 的内部工作机制是非常不同的。2.0 版本执行为 GPU 专门定制的程序。如果没有 GLKit 和 GLKBaseEffect 类，完成这个简单的例子就需要用 OpenGL ES 2.0 的“Shading Language”编写一个小的 GPU 程序。GLKBaseEffect 会在需要的时候自动地构建 GPU 程序并极大地简化本书中的例子。

控制渲染像素颜色的方式有很多种。这个应用的 GLKBaseEffect 实例使用一个恒定不变的白色来渲染三角形。这就意味着在三角形中的每一个像素都有相同的颜色。下面

的代码使用在 GLKit 中定义的用于保存 4 个颜色元素值的 C 数据结构体 GLKVector4 来设置这个恒定的颜色：

```
self.baseEffect.constantColor = GLKVector4Make(
   1.0f, // Red
   1.0f, // Green
   1.0f, // Blue
   1.0f);// Alpha

   // Set the background color stored in the current context
   glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // background color
```

前三个颜色元素是第 1 章图 1-2 中介绍的红、绿、蓝。第 4 个值为透明度，它决定了像素是半透明还是不透明。透明度元素会在第 3 章做详细介绍。设置红、绿、蓝为满值 1.0，以设置成白色。设置透明度为满值以使颜色为完全不透明。红、绿、蓝和透明度值统称为一个 RGBA 颜色。GLKVector4Make() 函数返回一个用指定的值初始化的 GLKit GLKVector4 结构体。

glClearColor() 函数设置当前 OpenGL ES 的上下文的“清除颜色”为不透明黑色。清除颜色由 RGBA 颜色元素值组成，用于在上下文的帧缓存被清除时初始化每个像素的颜色值。

第 1 章介绍了用于在 CPU 控制的内存和 GPU 控制的内存之间交换数据的缓存的概念。用于定义要绘制的三角形的顶点位置数据必须要发送到 GPU 来渲染。创建并使用一个用于保存顶点数据的顶点属性数组缓存。前 3 个步骤如下：

1）为缓存生成一个独一无二的标识符。

2）为接下来的运算绑定缓存。

3）复制数据到缓存中。

下面来自 -viewDidLoad 方法的实现的代码执行了前 3 步：

```
// Generate, bind, and initialize contents of a buffer to be
// stored in GPU memory
glGenBuffers(1,                 // STEP 1
   &vertexBufferID);
glBindBuffer(GL_ARRAY_BUFFER,  // STEP 2
   vertexBufferID);
glBufferData(                   // STEP 3
   GL_ARRAY_BUFFER,  // Initialize buffer contents
   sizeof(vertices), // Number of bytes to copy
   vertices,         // Address of bytes to copy
   GL_STATIC_DRAW);  // Hint: cache in GPU memory
```

在第 1 步中，glGenBuffers() 函数的第一个参数用于指定要生成的缓存标识符的数量，第二个参数是一个指针，指向生成的标识符的内存保存位置。在当前情况下，一个

标识符被生成，并保存在 vertexBufferID 实例变量中。

在第 2 步中，glBindBuffer() 函数绑定用于指定标识符的缓存到当前缓存。OpenGL ES 保存不同类型的缓存标识符到当前 OpenGL ES 上下文的不同部位。但是，在任意时刻每种类型只能绑定一个缓存。如果在这个例子中使用了两个顶点属性数组缓存，那么在同一时刻它们不能都被绑定。

glBindBuffer() 的第一个参数是一个常量，用于指定要绑定哪一种类型的缓存。OpenGL ES 2.0 对于 glBindBuffer() 的实现只支持两种类型的缓存，GL_ARRAY_BUFFER 和 GL_ELEMENT_ARRAY_BUFFER。GL_ELEMENT_ARRAY_BUFFER 将会在第 6 章详细解释。GL_ARRAY_BUFFER 类型用于指定一个顶点属性数组，例如本例中三角形顶点的位置。glBindBuffer() 的第二个参数是要绑定的缓存的标识符。

注意 缓存标识符实际上是无符号整型。0 值表示没有缓存。用 0 作为第二个参数调用 glBindBuffer() 函数来配置当前上下文的话，没有指定类型的缓存会被绑定。缓存标识符在 OpenGL ES 文档中又叫做“names”。

在第 3 步中，glBufferData 函数复制应用的顶点数据到当前上下文所绑定的顶点缓存中。

```
glBufferData(                    // STEP 3
GL_ARRAY_BUFFER,  // Initialize buffer contents
sizeof(vertices), // Number of bytes to copy
vertices,         // Address of bytes to copy
GL_STATIC_DRAW);  // Hint: cache in GPU memory
```

glBufferData() 的第一个参数用于指定要更新当前上下文中所绑定的是哪一个缓存。第二个参数指定要复制进这个缓存的字节的数量。第三个参数是要复制的字节的地址。最后，第 4 个参数提示了缓存在未来的运算中可能将会被怎样使用。GL_STATIC_DRAW 提示会告诉上下文，缓存中的内容适合复制到 GPU 控制的内存，因为很少对其进行修改。这个信息可以帮助 OpenGL ES 优化内存使用。使用 GL_DYNAMIC_DRAW 作为提示会告诉上下文，缓存内的数据会频繁改变，同时提示 OpenGL ES 以不同的方式来处理缓存的存储。

2. -glkView:drawInRect:

每当一个 GLKView 实例需要被重绘时，它都会让保存在视图的上下文属性中的 OpenGL ES 的上下文成为当前上下文。如果需要的话，GLKView 实例会绑定与一个 Core Animation 层分享的帧缓存，执行其他的标准 OpenGL ES 配置，并发送一个消息来调用 OpenGLES_Ch2_1ViewController 的 -glkView:drawInRect: 方法。-glkView:drawInRect: 是 GLKView 类的委托方法。作为 GLKViewController 的子类，OpenGLES_Ch2_1ViewController 会自动成为从 storyboard 文件加载的关联视图的委托。

下面委托方法的实现告诉 baseEffect 准备好当前 OpenGL ES 的上下文，以便为使用 baseEffect 生成的属性和 Shading Language 程序的绘图做好准备。接着，调用 OpenGL ES 的 glClear() 函数来设置当前绑定的帧缓存的像素颜色渲染缓存中的每一个像素的颜色为前面使用 glClearColor() 函数设定的值。正如 2.1 节所描述的，帧缓存可能有除了像素颜色渲染缓存之外的其他附加的缓存，并且如果其他的缓存被使用了，它们可以通过在 glClear() 函数中指定不同的参数来清除。glClear() 函数会有效地设置帧缓存中的每一个像素的颜色为背景色。

```
////////////////////////////////////////////////////////////////
// GLKView delegate method: Called by the view controller's view
// whenever Cocoa Touch asks the view controller's view to
// draw itself. (In this case, render into a Frame Buffer that
// shares memory with a Core Animation Layer)
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect
{
   [self.baseEffect prepareToDraw];

   // Clear Frame Buffer (erase previous drawing)
   glClear(GL_COLOR_BUFFER_BIT);

   // Enable use of currently bound vertex buffer
   glEnableVertexAttribArray(      // STEP 4
      GLKVertexAttribPosition);

   glVertexAttribPointer(          // STEP 5
      GLKVertexAttribPosition,
       3,                   // three components per vertex
       GL_FLOAT,            // data is floating point
       GL_FALSE,            // no fixed point scaling
       sizeof(SceneVertex), // no gaps in data
       NULL);               // NULL tells GPU to start at
                            // beginning of bound buffer

   // Draw triangles using the first three vertices in the
   // currently bound vertex buffer
   glDrawArrays(GL_TRIANGLES,      // STEP 6
      0,  // Start with first vertex in currently bound buffer
      3); // Use three vertices from currently bound buffer
}
```

在帧缓存被清除以后，是时候用存储在当前绑定的 OpenGL ES 的 GL_ARRAY_BUFFER 类型的缓存中的顶点数据绘制例子中的三角形了。使用缓存的前三步已经在 -viewDidLoad 方法中被执行了。正如第 1 章所描述的，OpenGLES_Ch2_1ViewController 的“glkView:drawInRect:”方法会执行剩下的几个步骤：

4）启动。

5）设置指针。

6）绘图。

在第 4 步中，通过调用 glEnableVertexAttrib-Array() 来启动顶点缓存渲染操作。OpenGL ES 所支持的每一个渲染操作都可以单独地使用保存在当前 OpenGL ES 上下文中的设置来开启或关闭。

在第 5 步中，glVertextAttribPointer() 函数会告诉 OpenGL ES 顶点数据在哪里，以及怎么解释为每个顶点保存的数据。在这个例子中，glVertexAttribPointer() 的第一个参数指示当前绑定的缓存包含每个顶点的位置信息。第二个参数指示每个位置有 3 个部分。第三个参数告诉 OpenGL ES 每个部分都保存为一个浮点类型的值。第四个参数告诉 OpenGL ES 小数点固定数据是否可以被改变。本书中没有例子会使用小数点固定的数据，因此这个参数值是 GL_FALSE。

注意 小数点固定类型是 OpenGL ES 支持的对于浮点类型的一种替代。小数点固定类型用牺牲精度的方法来节省内存。所有现代 GPU 都对浮点数的使用做了优化，并且小数点固定数据在使用之前最终都会被转换成浮点数。因此坚持使用浮点数可以减少 GPU 的运算量并提高精度。

第五个参数叫做“步幅”，它指定了每个顶点的保存需要多少个字节。换句话说，步幅指定了 GPU 从一个顶点的内存开始位置转到下一个顶点的内存开始位置需要跳过多少字节。sizeof（GLKVector3）指示在缓存中没有额外的字节，即顶点位置数据是密封的。在一个顶点缓存中保存除了每个顶点位置的 X、Y、Z 坐标之外的其他数据也是可能的。图 2-8 中的顶点数据内存模型显示了顶点存储器的一些选项。第一个图显示的是每个顶点的 3D 顶点位置坐标都紧密地保存在 12 字节中，就像 OpenGLES_Ch2_1 那个例子一样。第二个图显示的是用于每个顶点的存储的额外字节，因此在内存中在一个顶点与下一个顶点的位置坐标之间有缺口。

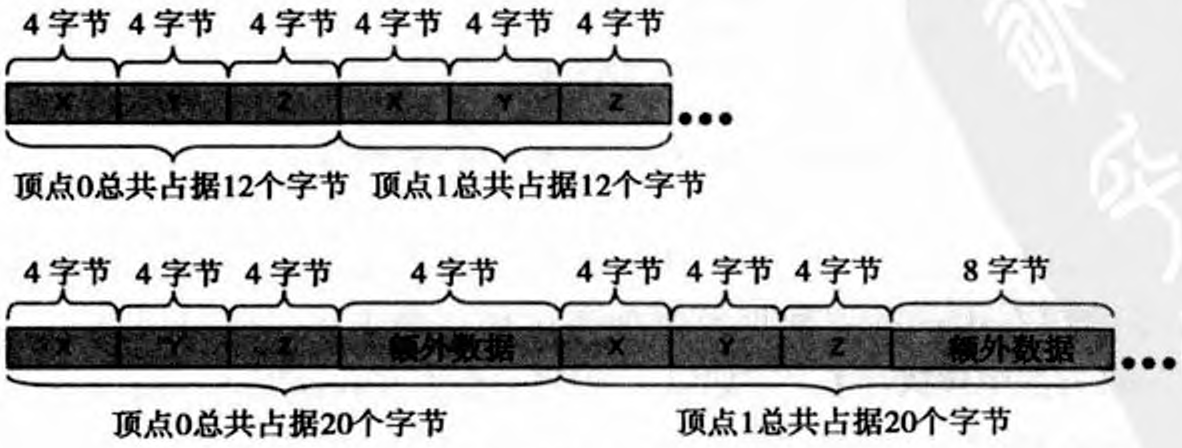


图 2-8 在顶点数组缓存内存中的顶点数据的一些可能的排列方式

glVertexAttribPointer() 的最后一个参数是 NULL，这告诉 OpenGL ES 可以从当前绑定的顶点缓存的开始位置访问顶点数据。

在第 6 步中，通过调用 glDrawArrays() 来执行绘图。glDrawArrays() 的第一个参数会告诉 GPU 怎么处理在绑定的顶点缓存内的顶点数据。这个例子会指示 OpenGL ES 去渲染三角形。glDrawArrays() 的第二个参数和第三个参数分别指定缓存内的需要渲染的第一个顶点的位置和需要渲染的顶点的数量。至此，在图 2-3 中显示的场景已经被完全地渲染出来或者至少在 GPU 处理完成后它就会被完全地渲染出来。请记住 GPU 运算与 CPU 运算是异步的。在这个例子中的所有代码都是运行在 CPU 上的，然后在需要进一步处理的时候向 GPU 发送命令。GPU 可能也会处理发送自 iOS 的 Core Animation 的命令，因此在任何给定的时刻 GPU 总共要执行多少处理并不一定。

3. -viewDidUnload

OpenGLES_Ch2_1ViewController 实现的最后一个方法是 -viewDidUnload 方法。和 -viewDidLoad 在与视图控制器相关的视图被加载时会被自动调用一样，-viewDidUnload 方法会在视图最终被卸载时调用。卸载的视图将不再被绘制，因此任何只是在绘制时需要的 OpenGL ES 缓存都可以被安全地删除。

第 7 步是删除不再需要的顶点缓存和上下文。设置 vertexBufferID 为 0 避免了在对应的缓存被删除以后还使用其无效的标识符。设置视图的上下文属性为 nil 并设置当前上下文为 nil，以便让 Cocoa Touch 收回所有上下文使用的内存和其他资源。

```
//////////////////////////////////////////////////////////////////
// Called when the view controller's view has been unloaded
// Perform clean-up that is possible when you know the view
// controller's view won't be asked to draw again soon.
- (void)viewDidUnload
{
   [super viewDidUnload];

   // Make the view's context current
   GLKView *view = (GLKView *)self.view;
   [EAGLContext setCurrentContext:view.context];

   // Delete buffers that aren't needed when view is unloaded
   if (0 != vertexBufferID)
   {
      glDeleteBuffers (1,          // STEP 7
                       &vertexBufferID);
      vertexBufferID = 0;
   }

   // Stop using the context created in -viewDidLoad
   ((GLKView *)self.view).context = nil;
   [EAGLContext setCurrentContext:nil];
}

@end
```

2.3.5 支持文件

在图 2-6 中显示的组内的 png 文件是 OpenGLES_Ch2_1 应用的图标。操作系统会根据应用所运行的设备是 iPhone、iPod Touch 还是 iPad 而自动地选择正确的图标。OpenGL_ES_for_iOS_72x72.png 文件包含 iPad 要用到的图标图像。OpenGL_ES_for_iOS_114x114.png 和 OpenGL_ES_for_iOS_57x57.png 包含 iPod Touch 和 iPhone 要用到的图标图像。扩展名 .png 代表 Portable Network Graphics（PNG）。 iOS 设备原生支持 PNG 文件并且会按照国际标准化组织的标准存储图像。

OpenGLES_Ch2_1-Info.plist 文件是在新工程创建的时候由 Xcode 自动生成的。OpenGLES_Ch2_1-Info.plist 保存诸如应用的版本号、使用的 storyboard 文件的名字、图标文件的名字等配置信息。添加应用特有的信息到这个文件是可能的，但是 OpenGLES_Ch2_1 应用不需要任何非标准的信息。配置文件只会在应用每次启动的时候由应用读取一次。

InfoPlist.strings 文件是在新工程创建的时候由 Xcode 自动生成的。InfoPlist.strings 文件包含在 OpenGLES_Ch2_1-Info.plist 文件中使用的字符串的本地化版本。本地化是指为不同的语言和文化团体提供专门的文本和图像。InfoPlist.strings 提供了一个根据用户的地区为相同的应用提供不同的图标和用户界面的方式。

1. main() 函数

main.m 文件是工程创建的时候由 Xcode 自动生成的并且包含应用的 main() 函数的实现。main() 函数被操作系统调用来开启 ANSI C、C++ 和 Objective-C 程序的运行。生成的 main.m 文件包含如下代码：

```
//
//  main.m
//  OpenGLES_Ch2_1
//


#import <UIKit/UIKit.h>

#import "OpenGLES_Ch2_1AppDelegate.h"

int main(int argc, char *argv[])
{
   @autoreleasepool
   {
       return UIApplicationMain(argc, argv, nil,
          NSStringFromClass([OpenGLES_Ch2_1AppDelegate class]));
   }
}
```

main() 函数使用 Objective-C 的 @autoreleasepool 关键字来开启自动引用计数[⊖] (Automatic Reference Counting)。main() 函数还会调用 UIApplicationMain() 函数，这个函数创建了包含 UIApplication 实例在内的应用的关键对象，同时开始处理用户事件。NSStringFromClass([OpenGLES_Ch2_AppDelegate class]) 表达式指定了与新创建的 UIApplication 实例一起使用的应用委托类的名字。UIApplication 创建了一个充满整个显示屏的 UIWindow，同时加载了一个或者多个 storyboard 文件来构建应用的用户界面。UIApplicationMain() 函数不会干涉 UIApplication 的执行，直到应用退出才会返回。

2. 预编译头文件

OpenGLES_Ch2_1-Prefix.pch 文件是由 Xcode 自动生成的并且改善了这个例子中应用的编译速度。扩展名 .pch 代表预编译头文件（pre-compiled header）。

3. Frameworks 和 Products

Frameworks 是 iOS 应用用到的系统库和资源。在 Xcode 文件列表中的 Frameworks 文件夹指定了应用使用了哪些框架。OpenGLES_Ch2_1 例子使用了 OpenGLES.framework、QuartzCore.framework、GLKit.framework、UIKit.framework、Foundation.framework 和 CoreGraphics.framework。

Products 文件夹包含编译 Xcode 工程中指定的应用时产生的结果。

2.4 深入探讨 GLKView 是怎么工作的

在介绍 iOS 5 的 GLKit 之前，对于每个开发者来说创建一个类似于 GLKView 的 UIView 的子类是必要的。苹果没有提供 GLKit 类的源代码，但是根据 OpenGL ES 可以推断出类似 GLKView 这样的类的主要功能可能的实现方式。本节剩下的部分和 OpenGLES_Ch2_2 这个例子会介绍 AGLKView 类以及它对于 GLKView 的部分重新实现。AGLKView 类不应该用在产品代码中，它仅仅是为了消除对于 GLKView、Core Animation 和 OpenGL ES 之间的交互的神秘感。不管从哪个方面来说，苹果优化的、测试的、面向未来的对于 GLKit 的实现都是最好的。如果你对于深入了解 GLKView 没有兴趣，可以跳到下一节。

注意　如果你不想在你的应用中使用苹果的 GLKit 类，你就需要使用 AGLKView 示例或者自己再创建一个相似的类。本书剩下的例子都假设使用了 GLKit。因此 AGLKView 不会出现在本书其他的任何例子中。

⊖　一个在 Cocoa Touch 应用中提供自动对象内存管理的编译器级语言功能。

在 OpenGLES_Ch2_2 例子中的 AGLKView 类继承自 Cocoa Touch 的 UIView 类，下面是这个类的接口声明，与苹果的 GLKView 类接口相似。

```
//
//  AGLKView.h
//  OpenGLES_Ch2_1
//

#import <UIKit/UIKit.h>
#import <OpenGLES/ES2/gl.h>
#import <OpenGLES/ES2/glext.h>

@class EAGLContext;
@protocol AGLKViewDelegate;


////////////////////////////////////////////////////////////////
// This subclass of the Cocoa Touch UIView class uses OpenGL ES
// to render pixel data into a Frame Buffer that shares pixel
// color storage with a Core Animation Layer.
@interface AGLKView : UIView
{
   EAGLContext   *context;
   GLuint        defaultFrameBuffer;
   GLuint        colorRenderBuffer;
   GLint         drawableWidth;
   GLint         drawableHeight;
}

@property (nonatomic, weak) IBOutlet id <AGLKViewDelegate> delegate;
@property (nonatomic, retain) EAGLContext *context;
@property (nonatomic, readonly) NSInteger drawableWidth;
@property (nonatomic, readonly) NSInteger drawableHeight;

- (void)display;

@end


#pragma mark - AGLKViewDelegate

@protocol AGLKViewDelegate <NSObject>

@required
- (void)glkView:(AGLKView *)view drawInRect:(CGRect)rect;

@end
```

AGLKViewDelegate 协议指定了一个任何 AGLKView 的委托都必须实现的方法。如果 AGLKView 实例的委托属性不等于 nil，每个 AGLKView 实例都会向它的委托发送“-glkView:drawInRect:”消息。

AGLKView 的实现比较简单易懂，但重写了来自 UIView 的多个方法并添加了一些用于支持 OpenGL ES 绘图的方法。

```
//
//  AGLKView.m
//  OpenGLES_Ch2_1
//

#import "AGLKView.h"
#import <QuartzCore/QuartzCore.h>


@implementation AGLKView

@synthesize delegate;
@synthesize context;

///////////////////////////////////////////////////////////////
// This method returns the CALayer subclass to be used by
// CoreAnimation with this view
+ (Class)layerClass
{
   return [CAEAGLLayer class];
}
```

每一个 UIView 实例都有一个相关联的被 Cocoa Touch 按需自动创建的 Core Animation 层。Cocoa Touch 会调用“+layerClass”方法来确定要创建什么类型的层。在这个例子中，AGLKView 类重写了继承自 UIView 的实现。当 Cocoa Touch 调用 AGLKView 实现的“+layerClass”方法时，它被告知要使用一个 CAEAGLLayer 类的实例，而不是原先的 CALayer。CAEAGLLayer 是 Core Animation 提供的标准层类之一。CAEAGLLayer 会与一个 OpenGL ES 的帧缓存共享它的像素颜色仓库。

接下来的代码块实现了“-initWithFrame:context:”方法并重写了继承来的“-initWithCoder:”方法。“-initWithFrame:context:”方法初始化了通过代码手动分配的实例。“-initWithCoder:”方法是 Cocoa Touch 用于初始化对象的标准方法之一。Cocoa Touch 会自动调用“-initWithCoder:”方法，这是反归档先前归档入一个文件的对象的过程的一部分。归档和反归档在其他流行的面向对象的框架中（比如 Java 和微软的 .NET）叫做串行化和反串行化。当 OpenGLES_Ch2_2 应用启动时，在这个例子中使用的 AGLKView 实例会自动地从应用的 storyboard 文件中加载（又叫做反归档）。

之后的两个方法的代码几乎是相同的。每个实例初始化的时候只有一个方法会被调

用。两个方法首先都会给超类 UIView 一个执行其初始化的机会，然后再执行这个例子需要的 Core Animation 和 OpenGL ES 上下文的一次性初始化。第一步是初始化视图的 Core Animation 层的本地指针，具体代码如下：

```
CAEAGLLayer *eaglLayer = (CAEAGLLayer *)self.layer;
```

需要C语言的类型转换（CAEAGLLayer *），这是因为UIView的“-layer”方法返回的是一个CALayer实例的指针。在AGLKView类的实现中，真正使用的是CAEAGLLayer类型，因此强制编译器接受CAEAGLLayer类型的这个转换是安全的。

```
//////////////////////////////////////////////////////////////////
// This method is designated initializer for the class
- (id)initWithFrame:(CGRect)frame context:(EAGLContext *)aContext;
{
   if ((self = [super initWithFrame:frame]))
   {
      CAEAGLLayer *eaglLayer = (CAEAGLLayer *)self.layer;

      eaglLayer.drawableProperties =
         [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithBool:NO],
            kEAGLDrawablePropertyRetainedBacking,
            kEAGLColorFormatRGBA8,
            kEAGLDrawablePropertyColorFormat,
            nil];

      self.context = aContext;
   }

   return self;
}


//////////////////////////////////////////////////////////////////
// This method is called automatically to initialize each Cocoa
// Touch object as the object is unarchived from an
// Interface Builder .xib or .storyboard file.
- (id)initWithCoder:(NSCoder*)coder
{
   if ((self = [super initWithCoder:coder]))
   {
      CAEAGLLayer *eaglLayer = (CAEAGLLayer *)self.layer;

      eaglLayer.drawableProperties =
         [NSDictionary dictionaryWithObjectsAndKeys:
            [NSNumber numberWithBool:NO],
            kEAGLDrawablePropertyRetainedBacking,
```

```
                kEAGLColorFormatRGBA8,
                kEAGLDrawablePropertyColorFormat,
                nil];
    }

    return self;
}
```

每个 AGLKView 的初始化方法会使用一个临时的 NSDictionary 实例来设置 eaglLayer 的 drawableProperties 属性。Dictionary 会保存键值对的集合。每个值都可以用对应的键快速获取。NSDictionary 是一个 Cocoa Touch 类，在这里被 CAEAGLLayer 类实例使用是为了保存层中用到的 OpenGL ES 的帧缓存类型的信息。

```
eaglLayer.drawableProperties =
    [NSDictionary dictionaryWithObjectsAndKeys:
        [NSNumber numberWithBool:NO],
        kEAGLDrawablePropertyRetainedBacking,
        kEAGLColorFormatRGBA8,
        kEAGLDrawablePropertyColorFormat,
        nil];
```

这个示例设置 kEAGLDrawablePropertyRetainedBacking 键的值为 NO 并设置 kEAGL-DrawablePropertyColorFormat 键的值为 kEAGLColorFormatRGBA8。不使用“保留背景”的意思是告诉 Core Animation 在层的任何部分需要在屏幕上显示的时候都要绘制整个层的内容。换句话说，这段代码是告诉 Core Animation 不要试图保留任何以前绘制的图像留作以后重用。RGBA8 颜色格式是告诉 Core Animation 用 8 位来保存层内的每个像素的每个颜色元素的值。

两个手动实现的访问器方法用于设置和返回视图的特定于平台的 OpenGL ES 上下文。因为 AGLKView 实例需要创建和配置一个帧缓存和一个像素颜色渲染缓存来与视图的 Core Animation 层一起使用，所以设置上下文会引起一些副作用。由于上下文保存缓存，因此修改视图的上下文会导致先前创建的所有缓存全部失效，并需要创建和配置新的缓存。

会受缓存操作影响的上下文是在调用 OpenGL ES 函数之前设定为当前上下文的。请注意在下面的代码中，创建帧缓存和渲染缓存会遵循一些适用于其他类型的缓存的相同的步骤，包括在 OpenGLES_Ch2_1 例子中的顶点数组缓存。一个新的步骤会调用 glFramebufferRenderbuffer() 函数来配置当前绑定的帧缓存以便在 colorRenderBuffer 中保存渲染的像素颜色。

```
//////////////////////////////////////////////////////////////////
// This method sets the receiver's OpenGL ES Context. If the
// receiver already has a different Context, this method deletes
// OpenGL ES Frame Buffer resources in the old Context and the
```

```
// recreates them in the new Context.
- (void)setContext:(EAGLContext *)aContext
{
   if(context != aContext)
   {  // Delete any buffers previously created in old Context
      [EAGLContext setCurrentContext:context];

      if (0 != defaultFrameBuffer)
      {
         glDeleteFramebuffers(1, &defaultFrameBuffer); // Step 7
         defaultFrameBuffer = 0;
      }

      if (0 != colorRenderBuffer)
      {
         glDeleteRenderbuffers(1, &colorRenderBuffer); // Step 7
         colorRenderBuffer = 0;
      }

      context = aContext;

      if(nil != context)
      {  // Configure the new Context with required buffers
         context = aContext;
         [EAGLContext setCurrentContext:context];

         glGenFramebuffers(1, &defaultFrameBuffer);    // Step 1
         glBindFramebuffer(                            // Step 2
            GL_FRAMEBUFFER,
            defaultFrameBuffer);

         glGenRenderbuffers(1, &colorRenderBuffer);    // Step 1
         glBindRenderbuffer(                           // Step 2
            GL_RENDERBUFFER,
            colorRenderBuffer);

         // Attach color render buffer to bound Frame Buffer
         glFramebufferRenderbuffer(
            GL_FRAMEBUFFER,
            GL_COLOR_ATTACHMENT0,
            GL_RENDERBUFFER,
            colorRenderBuffer);
      }
   }
}

////////////////////////////////////////////////////////////////
// This method returns the receiver's OpenGL ES Context
- (EAGLContext *)context
```

```
{
   return context;
}
```

下面的“-display”方法设置视图的上下文为当前上下文，告诉 OpenGL ES 让渲染填满整个帧缓存，调用视图的“-drawRect:”方法来实现用 OpenGL ES 函数进行真正的绘图，然后让上下文调整外观并使用 Core Animation 合成器把帧缓存的像素颜色渲染缓存与其他相关层混合起来。

```
//////////////////////////////////////////////////////////////////////
// Calling this method tells the receiver to redraw the contents
// of its associated OpenGL ES Frame Buffer. This method
// configures OpenGL ES and then calls -drawRect:
- (void)display;
{
   [EAGLContext setCurrentContext:self.context];
   glViewport(0, 0, self.drawableWidth, self.drawableHeight);

   [self drawRect:[self bounds]];

   [self.context presentRenderbuffer:GL_RENDERBUFFER];
}
```

glViewport() 函数可以用来控制渲染至帧缓存的子集，但是在这个例子中使用的是整个帧缓存。

如果视图的委托属性不是 nil，“-drawRect:”方法会调用委托的“-glkView:drawInRect:”方法。没有委托，AGLKView 什么都不会绘制。AGLKView 的子类可以通过重写继承的“-drawRect:”实现来绘图，即使是没有指定委托。“-glkView:drawInRect:”的参数是一个要被绘制的视图和一个覆盖整个视图范围的矩形。

```
//////////////////////////////////////////////////////////////////////
// This method is called automatically whenever the receiver
// needs to redraw the contents of its associated OpenGL ES
// Frame Buffer. This method should not be called directly. Call
// -display instead which configures OpenGL ES before calling
// -drawRect:
- (void)drawRect:(CGRect)rect
{
   if(self.delegate)
   {
      [self.delegate glkView:self drawInRect:[self bounds]];
   }
}
```

任何在接收到视图重新调整大小的消息时，Cocoa Touch 都会调用下面的 -layoutSubviews 方法。视图附属的帧缓存和像素颜色渲染缓存取决于视图的尺寸。视图会自

动地调整相关层的尺寸。上下文的“-renderbufferStorage:fromDrawable:”方法会调整视图的缓存的尺寸以匹配层的新尺寸。

```
//////////////////////////////////////////////////////////////////
// This method is called automatically whenever a UIView is
// resized including just after the view is added to a UIWindow.
- (void)layoutSubviews
{
   CAEAGLLayer   *eaglLayer = (CAEAGLLayer *)self.layer;

   // Initialize the current Frame Buffer's pixel color buffer
   // so that it shares the corresponding Core Animation Layer's
   // pixel color storage.
   [context renderbufferStorage:GL_RENDERBUFFER
      fromDrawable:eaglLayer];

   // Make the Color Render Buffer the current buffer for display
   glBindRenderbuffer(GL_RENDERBUFFER, colorRenderBuffer);

   // Check for any errors configuring the render buffer
   GLenum status = glCheckFramebufferStatus(
      GL_FRAMEBUFFER) ;

   if(status != GL_FRAMEBUFFER_COMPLETE)
   {
      NSLog(@"failed to make complete frame buffer object %x",
         status);
   }
}
```

“-drawableWidth”和“-drawableHeight”方法是它们各自的属性的访问器。它们被实现用来通过 OpenGL ES 的 glGetRenderbufferParameteriv() 方法获取和返回当前上下文的帧缓存的像素颜色渲染缓存的尺寸。

```
//////////////////////////////////////////////////////////////////
// This method returns the width in pixels of current context's
// Pixel Color Render Buffer
- (NSInteger)drawableWidth;
{
   GLint          backingWidth;

   glGetRenderbufferParameteriv(
      GL_RENDERBUFFER,
      GL_RENDERBUFFER_WIDTH,
      &backingWidth);

   return (NSInteger)backingWidth;
}
```

```
//////////////////////////////////////////////////////////////////////
// This method returns the height in pixels of current context's
// Pixel Color Render Buffer
- (NSInteger)drawableHeight;
{
   GLint          backingHeight;

   glGetRenderbufferParameteriv(
      GL_RENDERBUFFER,
      GL_RENDERBUFFER_HEIGHT,
      &backingHeight);

   return (NSInteger)backingHeight;
}
```

最后，在一个对象可以被回收时 Cocoa Touch 会自动地调用“-dealloc”方法，然后它的资源就会返回给操作系统。AGLKView 实现“-dealloc”方法是为了确保视图的上下文不再是当前上下文，其次是为了设置上下文属性为 nil。如果在属性变成 nil 之后，视图的上下文不再被使用了，那这个上下文也会被自动回收。

```
//////////////////////////////////////////////////////////////////////
// This method is called automatically when the reference count
// for a Cocoa Touch object reaches zero.
- (void)dealloc
{
   // Make sure the receiver's OpenGL ES Context is not current
   if ([EAGLContext currentContext] == context)
   {
      [EAGLContext setCurrentContext:nil];
   }

   // Deletes the receiver's OpenGL ES Context
   context = nil;
}

@end
```

关于 AGLKView 类就介绍这些内容。例子 OpenGLES_Ch2_2 还包含一个与 GLKit 的 GLKViewController 类相似的 AGLKViewController 类。除了在显示 GLKit 类的运行机制时使用了 AGLKView 和 AGLKViewController 类而不是 GLKit 类之外，OpenGLES_Ch2_2 和 OpenGLES_Ch2_1 是相同的。

与 GLKit 的 GLKViewController 类相似，AGLKViewController 使用一个 Core Animation CADisplayLink 对象来调度和执行与控制器相关联的视图的周期性的重绘。CADisplayLink 本质上是一个用于显示更新的同步计时器，它能够被设置用来在每个显示更新或者其他更新时发送一个消息。CADisplayLink 计时器的周期是以显示更新来计量的。

显示更新率通常是由嵌入设备的硬件决定的，它代表一个帧缓存的内容每秒最多能够被在屏幕上通过的像素显示出来的次数。因此来自 CADisplayLink 的消息为重新渲染一个场景提供了理想的触发器。渲染速度如果快于显示刷新率就是一种浪费，因为用户永远看不到两次显示刷新之间的额外的帧缓存的更新。

2.5 对于 GLKit 的推断

GLKit 类封装并简化了 Cocoa Touch 应用和 OpenGL ES 之间的常见交互。就像推测已存的 GLKit 类可能是怎么实现的是一种指导一样，遵循苹果的用来构建 GLKit 的范例来创建新类也是可能的。

本章的 OpenGLES_Ch2_1 例子使用了 GLKit，并且为了实现两个目的添加了对于 OpenGL ES 函数的直接调用，这两个目的是：清除帧缓存，以及使用一个顶点数组缓存来绘图。就像 GLKit 的 GLKView 封装了帧缓存和层管理，例子 OpenGLES_Ch2_3 重构 OpenGLES_Ch2_1 的可重用 OpenGL ES 代码为两个新类：AGLKContext 和 AGLKVertexAttribArrayBuffer。本书使用 AGLK 作为类和函数的前缀来代表是对于 GLKit 类的追加。GLKit 将来的版本可能包含与 AGLK 类相似的类。苹果通常不会预告对于其框架的增强，并且即使苹果最终实现了与本书中的 AGLK 类具有相似功能的类，也无法保证苹果会按相同的方式实现它。

AGLKContext 类是一个在例子 OpenGLES_Ch2_1 中使用的内建的 EAGLContext 类的简单子类。对于本例来说，AGLKContext 仅仅添加了一个 clearColor 属性和一个用来告诉 OpenGL ES 去设置在上下文的帧缓存中的每个像素颜色为 clearColor 的元素值的“-clear:”方法。

```
//
//  GLKContext.h
//  OpenGLES_Ch2_3
//

#import <GLKit/GLKit.h>

@interface AGLKContext: EAGLContext
{
   GLKVector4 clearColor;
}
@property (nonatomic, assign) GLKVector4
   clearColor;

- (void)clear:(GLbitfield)mask;

@end
```

AGLKContext 的实现实现了“-clear:”方法和用于 clearColor 属性的访问器。通过调用一个 OpenGL ES 的 glClearColor() 函数来设置 clearColor 属性。“-clear:”方法通过调用 glClear() 方法来实现。

```
//
//  GLKContext.m
//  OpenGLES_Ch2_3
//

#import "AGLKContext.h"

@implementation AGLKContext

//////////////////////////////////////////////////////////////////
// This method sets the clear (background) RGBA color.
// The clear color is undefined until this method is called.
- (void)setClearColor:(GLKVector4)clearColorRGBA
{
   clearColor = clearColorRGBA;

   NSAssert(self == [[self class] currentContext],
      @"Receiving context required to be current context");

   glClearColor(
      clearColorRGBA.r,
      clearColorRGBA.g,
      clearColorRGBA.b,
      clearColorRGBA.a);
}


//////////////////////////////////////////////////////////////////
// Returns the clear (background) color set via -setClearColor:.
// If no clear color has been set via -setClearColor:, the
// return clear color is undefined.
- (GLKVector4)clearColor
{
   return clearColor;
}
//////////////////////////////////////////////////////////////////
// This method instructs OpenGL ES to set all data in the
// current Context's Render Buffer(s) identified by mask to
// colors (values) specified via -setClearColor: and/or
// OpenGL ES functions for each Render Buffer type.
- (void)clear:(GLbitfield)mask
{
   NSAssert(self == [[self class] currentContext],
      @"Receiving context required to be current context");
```

```
    glClear(mask);
}

@end
```

GLKVertexAttribArrayBuffer 类封装了使用 OpenGL ES 2.0 的顶点属性数组缓存（或者简称“顶点缓存”）的所有 7 个步骤。这个函数减少了应用需要调用的 OpenGL ES 函数的数量。在后续章节的例子中重用 AGLKVertexAttribArrayBuffer 会在尽可能减少与 7 个缓存管理步骤相关的错误的同时减少编写的代码量。

```
//
//  GLKVertexAttribArrayBuffer.h
//  OpenGLES_Ch2_3
//

#import <GLKit/GLKit.h>

@class AGLKElementIndexArrayBuffer;


@interface AGLKVertexAttribArrayBuffer : NSObject
{
   GLsizeiptr   stride;
   GLsizeiptr   bufferSizeBytes;
   GLuint       glName;
}

@property (nonatomic, readonly) GLuint
   glName;
@property (nonatomic, readonly) GLsizeiptr
   bufferSizeBytes;
@property (nonatomic, readonly) GLsizeiptr
   stride;

- (id)initWithAttribStride:(GLsizeiptr)stride
   numberOfVertices:(GLsizei)count
   data:(const GLvoid *)dataPtr
   usage:(GLenum)usage;

- (void)prepareToDrawWithAttrib:(GLuint)index
   numberOfCoordinates:(GLint)count
   attribOffset:(GLsizeiptr)offset
   shouldEnable:(BOOL)shouldEnable;

- (void)drawArrayWithMode:(GLenum)mode
   startVertexIndex:(GLint)first
   numberOfVertices:(GLsizei)count;

@end
```

下面的实现在三个方法中封装了 7 个缓存管理步骤。AGLKVertextAttribArrayBuffer 类包含一些错误检查代码，但除此之外还包含一个对于例子 OpenGLES_Ch2_1 中的缓存管理代码的简单重用和重构。除了在类接口中声明的 3 个方法之外，还实现了一个“-dealloc”方法来删除一个相关联的 OpenGL ES 缓存标识符。

```
//
//  GLKVertexAttribArrayBuffer.m
//  OpenGLES_Ch2_3
//

#import "AGLKVertexAttribArrayBuffer.h"

@interface AGLKVertexAttribArrayBuffer ()

@property (nonatomic, assign) GLsizeiptr
   bufferSizeBytes;

@property (nonatomic, assign) GLsizeiptr
   stride;

@end


@implementation AGLKVertexAttribArrayBuffer

@synthesize glName;
@synthesize bufferSizeBytes;
@synthesize stride;


////////////////////////////////////////////////////////////////////////

// This method creates a vertex attribute array buffer in
// the current OpenGL ES context for the thread upon which this
// method is called.
- (id)initWithAttribStride:(GLsizeiptr)aStride
   numberOfVertices:(GLsizei)count
   data:(const GLvoid *)dataPtr
   usage:(GLenum)usage;
{
   NSParameterAssert(0 < aStride);
   NSParameterAssert(0 < count);
   NSParameterAssert(NULL != dataPtr);

   if(nil != (self = [super init]))
   {
      stride = aStride;
      bufferSizeBytes = stride * count;
```

```
        glGenBuffers(1,                    // STEP 1
           &glName);
        glBindBuffer(GL_ARRAY_BUFFER,  // STEP 2
           self.glName);
        glBufferData(                      // STEP 3
           GL_ARRAY_BUFFER,  // Initialize buffer contents
           bufferSizeBytes,  // Number of bytes to copy
           dataPtr,          // Address of bytes to copy
           usage);           // Hint: cache in GPU memory

        NSAssert(0 != glName, @"Failed to generate glName");
     }

     return self;
}


////////////////////////////////////////////////////////////////
// A vertex attribute array buffer must be prepared when your
// application wants to use the buffer to render any geometry.
// When your application prepares an buffer, some OpenGL ES state
// is altered to allow bind the buffer and configure pointers.
- (void)prepareToDrawWithAttrib:(GLuint)index
     numberOfCoordinates:(GLint)count
     attribOffset:(GLsizeiptr)offset
     shouldEnable:(BOOL)shouldEnable
{
     NSParameterAssert((0 < count) && (count < 4));
     NSParameterAssert(offset < self.stride);
     NSAssert(0 != glName, @"Invalid glName");
      glBindBuffer(GL_ARRAY_BUFFER,      // STEP 2
         self.glName);

      if(shouldEnable)
      {
         glEnableVertexAttribArray(      // Step 4
            index);
      }

      glVertexAttribPointer(             // Step 5
         index,                // Identifies the attribute to use
         count,                // number of coordinates for attribute
         GL_FLOAT,             // data is floating point
         GL_FALSE,             // no fixed point scaling
         self.stride,          // total num bytes stored per vertex
         NULL + offset);       // offset from start of each vertex to
                               // first coord for attribute
}
```

```
//////////////////////////////////////////////////////////////////
// Submits the drawing command identified by mode and instructs
// OpenGL ES to use count vertices from the buffer starting from
// the vertex at index first. Vertex indices start at 0.
- (void)drawArrayWithMode:(GLenum)mode
   startVertexIndex:(GLint)first
   numberOfVertices:(GLsizei)count
{
   NSAssert(self.bufferSizeBytes >=
      ((first + count) * self.stride),
      @"Attempt to draw more vertex data than available.");

   glDrawArrays(mode, first, count); // Step 6
}


//////////////////////////////////////////////////////////////////
// This method deletes the receiver's buffer from the current
// Context when the receiver is deallocated.
- (void)dealloc
{
    // Delete buffer from current context
    if (0 != glName)
    {
        glDeleteBuffers (1,             // STEP 7
                         &glName);
        glName = 0;
    }
}

@end
```

仔细检查例子 OpenGLES_Ch2_3 中的 OpenGLES_Ch2_3ViewController 类的实现，并把它与例子 OpenGLES_Ch2_1 中的 OpenGLES_Ch2_1ViewController 类的实现做下对比。在 OpenGLES_Ch2_3ViewController 中没有对于 OpenGL ES 函数的直接调用。另外，你可能会注意到一些小的代码风格的不一致。例如 OpenGLES_Ch2_1 使用一个 GLKVector4 结构体来设置恒定的颜色，但是用来设置清除颜色的对于 OpenGL ES 函数的调用可以直接接收 RGBA 颜色元素值：

```
self.baseEffect.constantColor = GLKVector4Make(
   1.0f, // Red
   1.0f, // Green
   1.0f, // Blue
   1.0f);// Alpha

// Set the background color stored in the current context
glClearColor(0.0f, 0.0f, 0.0f, 1.0f); // background color
```

AGLKVertexAttribArrayBuffer 和 AGLKContext 类扩展了 GLKit 引入的代码样式，因此 OpenGLES_Ch2_3 例子会使用一个一贯的样式：

```
self.baseEffect.constantColor = GLKVector4Make(
   1.0f, // Red
   1.0f, // Green
   1.0f, // Blue
   1.0f);// Alpha

// Set the background color stored in the current context
((AGLKContext *)view.context).clearColor = GLKVector4Make(
   0.0f, // Red
   0.0f, // Green
   0.0f, // Blue
   1.0f);// Alpha
```

细小的样式一致性上的改进会让应用代码更整洁，但更重要的是，在 Objective-C 类中封装 OpenGL ES 代码会让代码重用更容易。减少缓存被创建并使用时出现程序错误的机会。同时，可让使用多个缓存的应用需要编写、测试和调试的代码更少。

2.6 小结

所有的图形 iOS 应用都包含 Core Animation 层。所有的绘图都发生在层之上。Core Animation 合成器会混合当前应用与操作系统的层，从而在 OpenGL ES 的后帧缓存中产生最终的像素颜色。之后 Core Animation 合成器切换前后缓存以便把合成的内容显示到屏幕上。

原生 iOS 应用使用的是苹果的 Cocoa Touch 框架，并且由 Xcode IDE 组建而成。由 Xcode 生成的 main.m 文件包含建立 Cocoa Touch 应用结构和从 storyboard 文件中加载应用的用户界面的代码。一个标准的 Cocoa Touch 对象集合会提供标准 iOS 应用的所有功能。特定于某一个应用的功能是通过修改在标准应用结构中的应用委托或者根视图控制器对象来实现的。复杂的应用会向 Xcode 工程添加很多的额外文件。

使用 OpenGL ES 的 Cocoa Touch 应用要么使用 Core Animation 层配置一个与一个 OpenGL ES 的帧缓存分享内存的自定义 UIView，要么使用内建的 GLKView 类。苹果的 GLKit 框架提供了 GLKView 以处理细节，因此开发者很少会重新创建 UIView 的子类。一个 OpenGL ES 的上下文会存储当前 OpenGL ES 的状态，并控制 GPU 硬件。每个 GLKView 实例都需要一个上下文。

本章中的 OpenGLES_Ch2_1 例子为接下来的例子奠定了基础。所有本例专有的代码都是在 OpenGLES_Ch2_1ViewController 类中实现的。应用的 storyboard 文件指定了 GLKView 实例，这个实例使用根视图控制器作为委托。委托会接收来自其他对象的消

息并在响应中执行特定应用的操作或者控制其他对象。

例子中 OpenGLES_Ch2_1ViewController 类的实现包含 3 个方法：一个会在视图从 storyboard 文件中加载时被自动调用，一个会在每次视图需要被重绘时被自动调用，还有一个会在视图卸载时被自动调用。这三个方法分别通过创建必要的上下文和顶点缓存来初始化 OpenGL ES，使用上下文和顶点缓存在 GLKView 的与 Core Animation 层分享内存的帧缓存中绘图，并删除上下文和顶点缓存。

GLKBaseEffect 类隐藏了 iOS 所支持的 OpenGL ES 版本之间的很多不同。当使用 OpenGL ES 2.0 时，GLKBaseEffect 会生成直接在 GPU 上运行的 Shading Language 程序。GLKBaseEffect 使程序员专注于应用的功能和图形概念，而无须学习 Shading Language。

本章中的 OpenGLES_Ch2_3 例子把 OpenGLES_Ch2_1 例子中的 OpenGL ES 函数调用重构为 AGLKVertexAttribArrayBuffer 类和 AGLKContext 类。

第 3 章会扩展 OpenGLES_Ch2_1 示例以使三角形渲染更加有趣和强大。

第3章 纹　理

第 2 章的例子演示了怎么使用 OpenGL ES 2.0 渲染和显示一个白色的三角形。现实中的对象很少仅由单一的白色或者其他固定的颜色组成。OpenGL ES 提供了一个为几何图形中的每个顶点设置不同颜色的方法，不过实际上即使是拥有多重颜色顶点的三角形也无法在渲染的场景中提供逼真的感觉。本章会介绍一个叫做纹理的图形技术，它可以控制一个渲染的三角形中的每个像素的颜色。

纹理是计算机图形中最复杂的话题之一。本章会介绍纹理的底层概念和常用选项。多个例子会逐步说明这些技术。

3.1　什么是纹理

纹理是一个用来保存图像的颜色元素值的 OpenGL ES 缓存。图 3-1 显示的是本章例子中的一个纹理使用的一个图像。纹理可以使用任何图像，包括树木、面孔、砖块、云彩，或者机器。当把纹理应用到几何图形中后，会使渲染的场景显得更自然，会使三角形的复杂组合像是真实的物体而不只是有颜色的面。

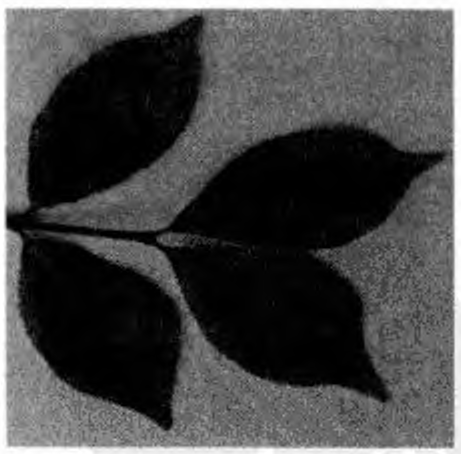


图 3-1　一个用作 OpenGL ES 纹理的图像

在纹理的缓存中保存的颜色值可能要耗费很多的内存。所有的嵌入式系统都为纹理设定了内存的最大尺寸限制。所有的 iPhone 和 iPad Touch 版本都可以支持由 1024×1024 像素的图像生成的纹理。iPad 和最新一代 iPhone、iPod Touch 所支持的尺寸甚至更大。但是，由于嵌入式系统的可用内存相对较小，应尽量使用最小的图像来产生可以接受的渲染结果。

当用一个图像初始化一个纹理缓存之后，在这个图像中的每个像素变成了纹理中的一个纹素（texel）。与像素类似，纹素保存颜色数据。像素和纹素之间的差别很微妙：像素通常表示计算机屏幕上的一个实际的颜色点，因此，像素通常被用来作为一个测量单位。说一个图像有 256 像素宽，256 像素高是正确的。与此相反，纹素存在于一个虚拟

的没有尺寸的数学坐标系中。图 3-2 显示了 OpenGL ES 纹理坐标系中的一个纹理。

纹理坐标系有一个命名为 S 和 T 的 2D 轴。在一个纹理中无论有多少个纹素，纹理的尺寸永远是在 S 轴上从 0.0 到 1.0，在 T 轴上从 0.0 到 1.0。从一个 1 像素高 64 像素宽的图像初始化来的纹理会沿着整个 T 轴有 1 纹素，沿着 S 轴有 64 纹素。

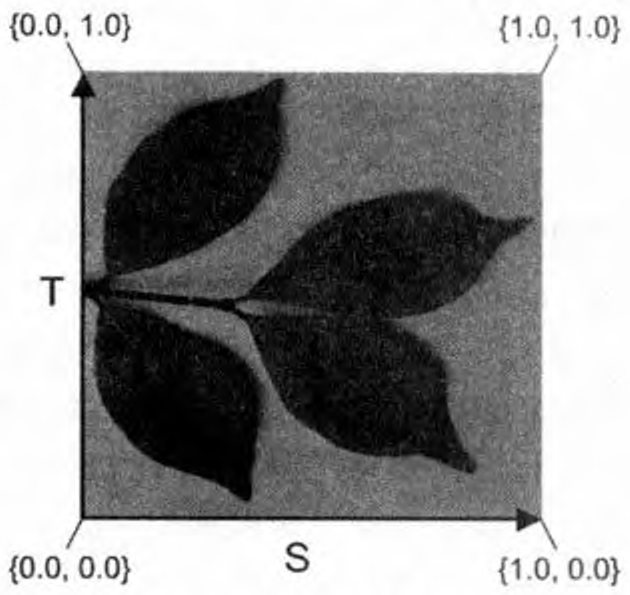


图 3-2 OpenGL ES 纹理坐标系

注意 本书中的例子使用的是 2D 纹理，但是一些 OpenGL 实现还会支持 1D 和 3D 纹理。一个 1D 的纹理就相当于沿着 T 轴只有 1 纹素的 2D 纹理，因此一个 64 纹素的 1D 纹理与从尺寸为 64×1 的图像初始化来的 2D 纹理是相同的。3D 纹理就像一个层饼，一个在拥有 R、S 和 T 轴的坐标系中沿着 R 轴堆叠的多个 2D 纹理的层饼。1D 和 3D 纹理对于某些特定的应用来说是非常方便的，但是 2D 纹理是迄今为止最常见的。

3.1.1 对齐纹理和几何图形

我们需要告诉 OpenGL ES 如何使用一个纹理来给几何图形对象着色。图 3-3 显示的是第 2 章例子中的简单三角形，但是这次它是用一个纹理绘制的。在图 3-3 中需要注意的第一件事是绘制的三角形的高度要比宽度大。这个三角形是用 {-0.5f,-0.5f,0.0}、{0.5f,-0.5f,0.0}、{-0.5f,0.5f,0.0} 这三个顶点定义的，这使得三角形的高度和宽度在纯数学的 OpenGL ES 坐标系中是相等的。但是这个例子的帧缓存是按像素来匹配屏幕尺寸的。在渲染时，GPU 会转换纯数学 OpenGL ES 坐标系中的每个顶点的 X、Y、Z 坐标为帧缓存中所对应的真实像素位置。帧缓存中的像素位置叫做视口（viewport）坐标。第 2 章曾简短地提到过视口。转换为视口坐标的结果是所有绘制的几何图形都被拉伸以适合屏幕大小，也就是说高比宽大了。第 5 章会讲解怎么控制 GPU 的这个转换。

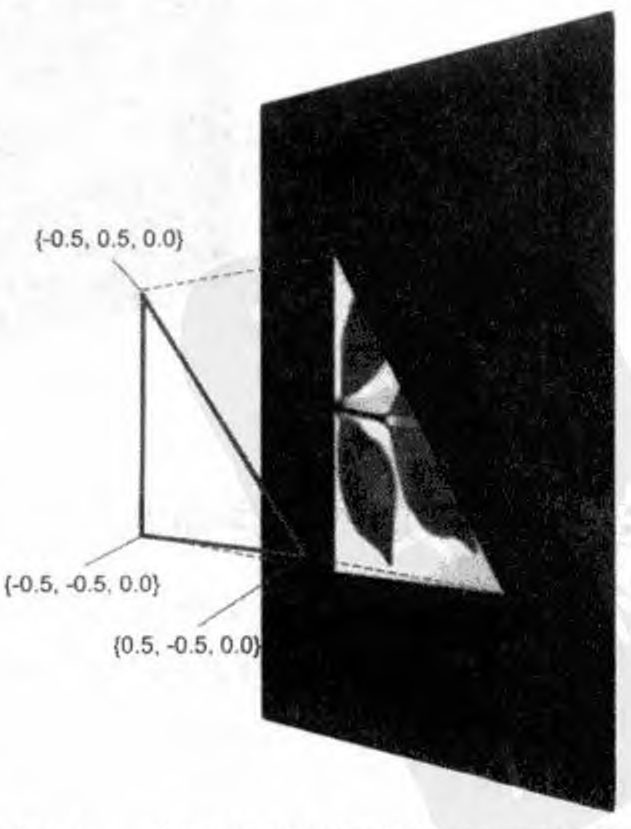


图 3-3 一个渲染进帧缓存的纹理三角形

注意 一个场景中的每个几何坐标的转换都要耗费一打的矢量数学运算，但是 GPU 经常会在每秒执行 10 亿次矢量运算。与内存读取相比这个转换是如此之快，以至于这个转换在实际的应用中几乎不会产生性能影响。

在每个顶点的 X、Y、Z 坐标被转换成视口坐标后，GPU 会设置转换生成的三角形内的每个像素的颜色。转换几何形状数据为帧缓存中的颜色像素的渲染步骤叫做点阵化（rasterizing），每个颜色像素叫做片元（fragment）。当 OpenGL ES 没有使用纹理时，GPU 会根据包含该片元的对象的顶点的颜色来计算每个片元的颜色。当设置了使用纹理后，GPU 会根据在当前绑定的纹理缓存中的纹素来计算每个片元的颜色。

程序需要指定怎么对齐纹理和顶点，以便让 GPU 知道每个片元的颜色由哪些纹素决定。这个对齐又叫做映射（mapping），是通过扩展为每个顶点保存的数据来实现的：除了 X、Y、Z 坐标，每个顶点还给出了 U 和 V 坐标值。每个 U 坐标会映射顶点在视口中的最终位置到纹理中的沿着 S 轴的一个位置。V 坐标映射到 T 轴。图 3-4 形象化了 3.2 节将介绍的 OpenGLES_Ch3_1 例子中演示的纹理映射。

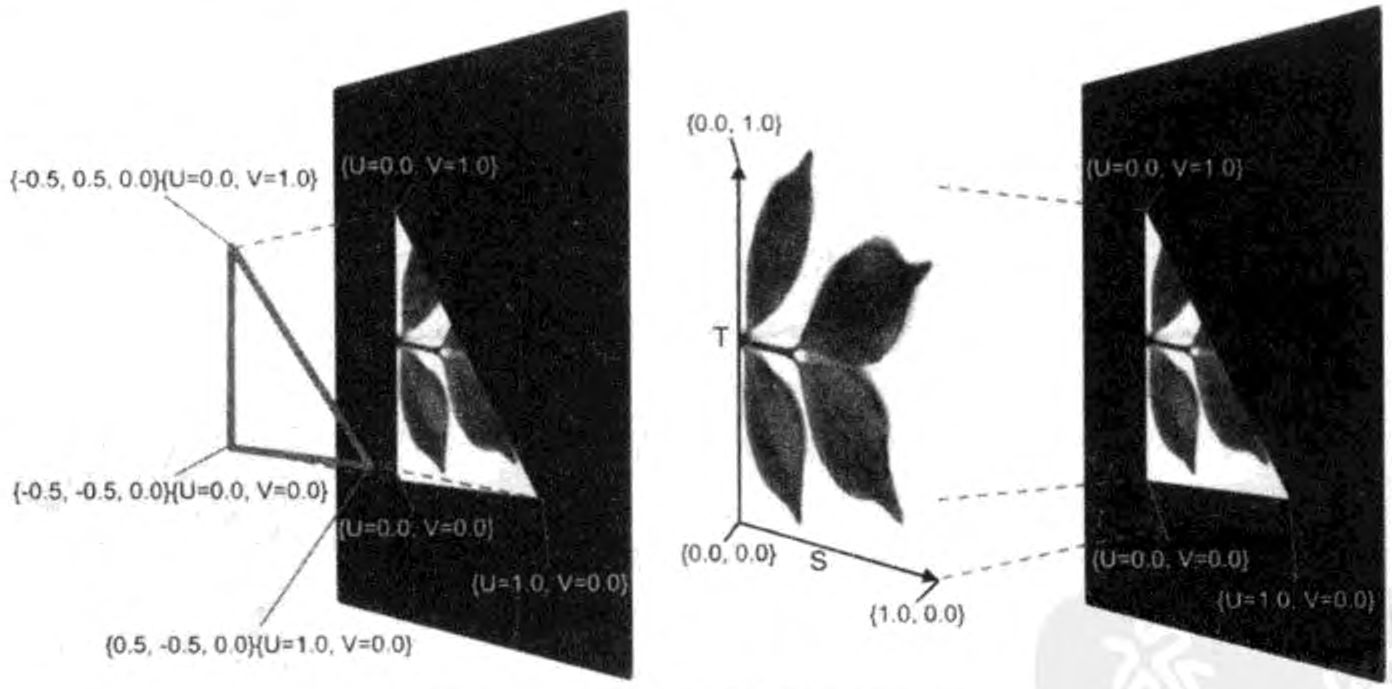


图 3-4 纹理映射到转换后的顶点

3.1.2 纹理的取样模式

每个顶点的 U 和 V 坐标会附加到每个顶点在视口坐标中的最终位置。然后 GPU 会根据计算出来的每个片元的 U、V 位置从绑定的纹理中选择纹素。这个选择过程叫做取样（Sampling）。取样会把纹理的 S 和 T 坐标系与每个渲染的三角形的顶点的 U、V 坐标匹配起来。如图 3-4 所示，在 S 和 T 坐标系中与 {0，0} 位置最近的纹素会被映射到拥有 {0，0} 的 U、V 坐标的顶点所对应的片元上。每个随后的片元位置对应于一个沿

着 S 和 T 轴的与该片元在 U、V 坐标中的位置等比例的位置。例如，一个 U、V 坐标为 {0.5,0,5} 的片元会被当前绑定的纹理中最接近中间位置的纹素所着色。

注意 {S,T} 坐标和 {U,V} 坐标通常是由程序员和 3D 设计师设定的。两者之间只有一个实际的差异：{U,V} 坐标可能会超出 0.0 到 1.0 的这个范围。栅格化会基于一个本节后面要介绍的可配置的“循环模式”映射 {S，T} 坐标为超出 0.0 到 1.0 范围的 {U,V} 坐标。

渲染过程中的取样可能会导致纹理被拉伸、压缩，甚至翻转。例子 OpenGLES_Ch3_3 会演示一些可能会产生的纹理变形。

OpenGL ES 支持多个不同的取样模式：考虑一下当一个三角形产生的片元少于绑定的纹理内的可用纹素的数量时会发生什么。一个拥有大量纹素的纹理被映射到帧缓存内的一个只覆盖几个像素的三角形中，这种情况会在任何时间发生。相反的情况也会发生。一个包含少量纹素的纹理可能会被映射到一个在帧缓存中产生很多个片元的三角形。程序会使用如下的 glTexParameteri() 函数来配置每个绑定的纹理，以便使 OpenGL ES 知道怎么处理可用纹素的数量与需要被着色的片元的数量之间的不匹配。

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

使用值 GL_LINEAR 来指定参数 GL_TEXTURE_MIN_FILTER 会告诉 OpenGL ES 无论何时出现多个纹素对应一个片元时，从相配的多个纹素中取样颜色，然后使用线性内插法来混合这些颜色以得到片元的颜色。产生的片元颜色可能最终是一个纹理中不存在的颜色。例如，一个纹理是由交替的黑色和白色纹素组成的，线性取样会混合纹素的颜色，因此片元最终会是灰色的。使用 GL_NEAREST 值来指定 GL_TEXTURE_MIN_FILTER 参数会产生一个不同的结果。与片元的 U、V 坐标最接近的纹素的颜色会被取样。如果一个纹理是由交替的黑色和白色纹素组成的，GL_NEAREST 取样模式会拾取其中的一个纹素或者另一个，并且最终的片元要么是白色的，要么是黑色的。下一节（包括图 3-5）会展示线性插值和最邻近差值取样模式之间的效果对比。

glTexParameteri() 的 GL_TEXTURE_MAG_FILTER 参数用于在没有足够的可用纹素来唯一性地映射一个或者多个纹素到每个片元上时配置取样。在这种情况下，GL_LINEAR 值会告诉 OpenGL ES 混合附近纹素的颜色来计算片元的颜色。GL_LINEAR 值会有一个放大纹理的效果，并会让它模糊地出现在渲染的三角形上。GL_TEXTURE_MAG_FILTER 的 GL_NEAREST 值仅仅会拾取与片元的 U、V 位置接近的纹素的颜色，并放大纹理，这会使它有点像素化地出现在渲染的三角形上。

除了减小和放大过滤选项，当 U、V 坐标的值小于 0 或者大于 1 时，程序会指定要发生什么。有两个选择，要么尽可能多地重复纹理以填满映射到几何图形的整个 U、V 区域，要么每当片元的 U、V 坐标的值超出纹理的 S、T 坐标系的范围时，取样纹理边缘的纹素。纹理的循环模式是为 S 和 T 轴分别设置的，参考下面的代码：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

本章的例子 OpenGLES_Ch3_3 会演示一些由这些纹理参数值产生的效果。

3.1.3 MIP 贴图

MIP 贴图是与取样密切相关的。根据维基百科，MIP 代表拉丁短语 Multum In Parvo，意思是“放置很多东西的小空间”。回忆一下，内存存取是现代图形处理的薄弱环节。当有多个纹素对应一个片元时，线性取样会导致 GPU 仅仅为了计算一个片元的最终颜色而读取多个纹素的颜色值。MIP 贴图是一个为纹理存储多个细节级别的技术。高细节的纹理会沿着 S 轴和 T 轴存储很多的纹素。低细节的纹理沿着每个轴存储很少的纹素。最低细节的纹理只保存一个纹素。多个细节级别增加了在 S、T 轴上的纹素和每个片元的 U、V 坐标之间有紧密的对应关系的可能性。当存在一个紧密的对应关系时，GPU 会减少取样纹素的数量，进而会减少内存访问的次数。

使用 MIP 贴图通常会通过减少 GPU 取样的数量来提高渲染的性能，但是 MIP 贴图使每个纹理所需要的内存增加了 1/3，在网页 http://en.wikipedia.org/wiki/Mipmap 上做了解释。在 iOS 设备上，内存限制可能优先于渲染性能。通常最好的策略是逐个在使用和不使用 MIP 贴图的情况下测试你的应用，以决定哪种方式的效果最好。

3.2 OpenGLES_Ch3_1 示例

纹理涉及很多潜在的复杂选项，但是苹果的 GLKit 极大地简化了常见纹理的配置。本例是基于第 2 章的 OpenGLES_Ch2_3 示例建立的。在 OpenGLES_Ch3_1 中的一个单独的文件，OpenGLES_Ch3_1ViewController 类的实现文件，包含了对 OpenGLES_Ch2_3 的所有修改代码。下面的代码块用粗体标识了变化部分。首先，纹理坐标被加入到 SceneVertex 类型的声明中：

```
//////////////////////////////////////////////////////////////////
// This data type is used to store information for each vertex
typedef struct {
   GLKVector3  positionCoords;
```

```
    GLKVector2  textureCoords;
}
SceneVertex;
```

纹理坐标定义了几何图形中的每个顶点的纹理映射。为例子保存顶点数据的vertices数组初始化了纹理坐标和位置坐标。

```
//////////////////////////////////////////////////////////////////
// Define vertex data for a triangle to use in example
static const SceneVertex vertices[] =
{
   {{-0.5f, -0.5f, 0.0f}, {0.0f, 0.0f}}, // lower left corner
   {{ 0.5f, -0.5f, 0.0f}, {1.0f, 0.0f}}, // lower right corner
   {{-0.5f,  0.5f, 0.0f}, {0.0f, 1.0f}}, // upper left corner
};
```

“-viewDidLoad”方法扩展了示例OpenGLES_Ch2_3中的代码来影响GLKit提供的GLKTextureLoader类，这个类用于将一个纹理图像加载到一个OpenGL ES纹理缓存中。

```
//////////////////////////////////////////////////////////////////
// Called when the view controller's view is loaded
// Perform initialization before the view is asked to draw
- (void)viewDidLoad
{
[super viewDidLoad];

// Verify the type of view created automatically by the
// Interface Builder storyboard
GLKView *view = (GLKView *)self.view;
NSAssert([view isKindOfClass:[GLKView class]],
   @"View controller's view is not a GLKView");

// Create an OpenGL ES 2.0 context and provide it to the
// view
view.context = [[AGLKContext alloc]
   initWithAPI:kEAGLRenderingAPIOpenGLES2];

// Make the new context current
[AGLKContext setCurrentContext:view.context];

// Create a base effect that provides standard OpenGL ES 2.0
// shading language programs and set constants to be used for
// all subsequent rendering
self.baseEffect = [[GLKBaseEffect alloc] init];
self.baseEffect.useConstantColor = GL_TRUE;
self.baseEffect.constantColor = GLKVector4Make(
   1.0f, // Red
   1.0f, // Green
```

```
        1.0f, // Blue
        1.0f);// Alpha

    // Set the background color stored in the current context
    ((AGLKContext *)view.context).clearColor = GLKVector4Make(
        0.0f, // Red
        0.0f, // Green
        0.0f, // Blue
        1.0f);// Alpha

    // Create vertex buffer containing vertices to draw
    self.vertexBuffer = [[AGLKVertexAttribArrayBuffer alloc]
        initWithAttribStride:sizeof(SceneVertex)
        numberOfVertices:sizeof(vertices) / sizeof(SceneVertex)
        data:vertices
        usage:GL_STATIC_DRAW];

    // Setup texture
    CGImageRef imageRef =
        [[UIImage imageNamed:@"leaves.gif"] CGImage];

    GLKTextureInfo *textureInfo = [GLKTextureLoader
        textureWithCGImage:imageRef
        options:nil
        error:NULL];

        self.baseEffect.texture2d0.name = textureInfo.name;
        self.baseEffect.texture2d0.target = textureInfo.target;
    }
```

CGImageRef 是一个在苹果的 Core Graphics 框架中定义的 C 数据类型。Core Graphics 包含很多强大的 2D 图像处理和绘制函数。UIImage 的“+imageNamed:”方法会返回一个初始化自图形文件的 UIImage 实例。很多不同的图像文件格式都支持。命名的图像必须被包含为应用的一部分，以便“+imageNamed:”可以找到它。

GLKTextureLoader 的“-textureWithCGImage:options:error:”方法会接受一个 CGImageRef 并创建一个新的包含 CGImageRef 的像素数据的 OpenGL ES 纹理缓存。这个方法看似强大。接受一个 CGImageRef 使得图像数据的源可以是任何 Core Graphics 支持的形式，从一个电影的单个帧到由一个应用绘制的自定义 2D 图像，再到一个图像文件的内容。“options:”参数接受一个存储了用于指定 GLKTextureLoader 怎么解析加载的图像数据的键值对的 NSDictionary。可用选项之一是指示 GLKTextureLoader 为加载的图像生成 MIP 贴图。

GLKTextureLoader 会自动调用 glTexParameteri() 方法来为创建的纹理缓存设置 OpenGL ES 取样和循环模式。如果使用了 MIP 贴图，并且 GL_TEXTURE_MIN_FILTER 被设置成 GL_LINEAR_MIPMAP_LINEAR，这会告诉 OpenGL ES 使用与被取样的 S、T 坐标最接近的纹素的线性插值取样两个最合适的 MIP 贴图图像尺寸

（细节级别）。然后，来自 MIP 贴图的两个样本被线性差值来产生最终的片元颜色。GL_LINEAR_MIPMAP_LINEAR 过滤器通常会产生高质量的渲染输出，但是会比其他模式需要更多的 GPU 计算。如果没有使用 MIP 贴图，GLKTextureLoader 会自动设置 GL_TEXTURE_MIN_FILTER 为 GL_LINEAR。GL_TEXTURE_WRAP_S 和 GL_TEXTURE_WRAP_T 都会被设置为 GL_CLAMP_TO_EDGE。

第 2 章例子中引入的 GLKBaseEffect 对象提供了对于使用纹理做渲染的内建的支持。在 OpenGL ES 纹理缓存被创建以后，这个例子设置 baseEffect 的 texture2d0 属性使用一个新的纹理缓存。

GLKTextureInfo 类封装了与刚创建的纹理缓存相关的信息，包括它的尺寸以及它是否包含 MIP 贴图，但是这个例子只需要缓存的 OpenGL ES 标识符、名字和用于纹理的 OpenGL ES 目标。OpenGL ES 上下文会为各种缓存分别保存配置信息。帧缓存会被单独从顶点属性数组缓存或者纹理缓存配置。事实上，OpenGL ES 的上下文支持多种纹理缓存。例如，一种纹理缓存会包含普通的 2D 图像数据，然后另一种会保存一个用于特殊效果的特别形状的图像，这些内容会在第 8 章中介绍。GLKTextureInfo 的 target 属性指定被配置的纹理缓存的类型。一些 OpenGL ES 实现还会为 1D 和 3D 纹理保持独立的纹理缓存目标。

```
//////////////////////////////////////////////////////////////////
// GLKView delegate method: Called by the view controller's view
// whenever Cocoa Touch asks the view controller's view to
// draw itself. (In this case, render into a frame buffer that
// shares memory with a Core Animation Layer)
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect
{
   [self.baseEffect prepareToDraw];

   // Clear back frame buffer (erase previous drawing)
   [(AGLKContext *)view.context clear:GL_COLOR_BUFFER_BIT];

   [self.vertexBuffer prepareToDrawWithAttrib:GLKVertexAttribPosition
      numberOfCoordinates:3
      attribOffset:offsetof(SceneVertex, positionCoords)
      shouldEnable:YES];
   [self.vertexBuffer prepareToDrawWithAttrib:GLKVertexAttribTexCoord0
      numberOfCoordinates:2
      attribOffset:offsetof(SceneVertex, textureCoords)
      shouldEnable:YES];

   // Draw triangles using the first three vertices in the
   // currently bound vertex buffer
   [self.vertexBuffer drawArrayWithMode:GL_TRIANGLES
      startVertexIndex:0
      numberOfVertices:3];
}
```

在 OpenGLES_Ch2_3 例子中创建并在这里重用的 AGLKVertexAttribArrayBuffer 类隐式地支持每个顶点的属性的任意组合。OpenGLES_Ch3_1ViewController 的“-glkView:drawInRect:”方法的实现首先告诉 vertexBuffer 让 OpenGL ES 为渲染顶点位置做好准备（就像例子 OpenGLES_Ch2_3），然后添加第二个对于“-prepareToDraw-WithAttrib:numberOfCoordinates:attribOffset:shouldEnable:”方法的调用。第二个调用告诉 vertexBuffer 让 OpenGL ES 为每个顶点的两个纹理坐标的渲染做好准备。编译器会用纹理坐标开始的每个 SceneVertex 结构体内的内存偏移来代替 ANSI C 的 offsetof() 宏。

在一个纹理被赋给 baseEffect，同时 OpenGL ES 为使用位置和纹理坐标属性做好准备之后，调用 AGLKVertexAttribArrayBuffer 的“-drawArrayWithMode:startVertexIndex:numberOfVertices:”方法来指示 OpenGL ES 去渲染有纹理的三角形。

例子 OpenGLES_Ch3_1 对于“-viewDidUnload”方法的实现与例子 OpenGLES_Ch2_3 的实现保持不变。

3.3 深入探讨 GLKTextureLoader 是怎么工作的

OpenGL ES 的纹理缓存与第 1 章讨论过的其他的缓存具有相同的步骤。首先使用 glGenTextures() 函数生成一个纹理缓存标识符；然后使用 glBindTexture() 函数将其绑定到当前上下文；接下来，通过使用 glTexImage2D() 函数复制图像数据来初始化纹理缓存的内容。例子 OpenGLES_Ch3_2 显示了所有的步骤。

大部分的 OpenGL ES 实现要么需要，要么受益于使用尺寸为 2 的幂的纹理。图 3-1 中的图像为 256×256 像素。这个尺寸符合 OpenGL ES 的要求，因为 256 是 2 的幂。2 的幂包括 $2^0 = 1$、$2^1 = 2$、$2^2 = 4$、$2^3 = 8$、$2^4 = 16$、$2^5 = 32$、$2^6 = 64$、$2^7 = 128$、$2^8 = 256$ 和 $2^9 = 512$。一个 4×64 的纹理是有效的，一个 128×128 的纹理可以工作良好，一个 1×64 的纹理也可以。一个 200×200 的纹理要么不工作，要么根据使用的 OpenGL ES 版本在渲染时导致效率低下。限制纹理的尺寸通常不会引起任何问题。例子 OpenGLES_Ch3_2 显示了怎么生成纹理缓存，按需重新调整图像的尺寸为 2 的幂，使用图像初始化纹理缓存。

示例 OpenGLES_Ch3_2 中的 AGLKTextureLoader 是苹果的 GLKit 的 GLKTexture-Loader 类的部分实现。AGLKTextureLoader 类不会用在产品代码中，它仅仅是为了消除关于 GLKTextureLoader、Core Graphcis 和 OpenGL ES 之间交互的神秘感。

使用 AGLKTextureLoader 代替 GLKit 的 GLKTextureLoader 是例子 OpenGLES_Ch3_2 中的 OpenGLES_Ch3_2ViewController 与例子 OpenGLES_Ch3_1 中的对应的视图控制器之间的唯一不同。在例子 OpenGLES_Ch3_2 中的 AGLKTextureLoader 类的接口和实现提供了 GLKit 提供的功能的一个子集。具体来说，GLKit 的 GLKTextureLoader

类支持异步纹理加载，MIP贴图生成，以及比简单的2D平面更加吸引人的纹理缓存类型。AGLKTextureLoader只复制了在例子OpenGLES_Ch3_1中用到的GLKTextureLoader的功能。

例子OpenGLES_Ch3_2中的AGLKTextureLoader.h文件声明了两个类：AGLKTextureInfo和AGLKTextureLoader。

```
//
//  AGLKTextureLoader.h
//  OpenGLES_Ch3_2
//

#import <GLKit/GLKit.h>

#pragma mark -AGLKTextureInfo

@interface AGLKTextureInfo : NSObject
{
@private
   GLuint name;
   GLenum target;
   GLuint width;
   GLuint height;
}

@property (readonly) GLuint name;
@property (readonly) GLenum target;
@property (readonly) GLuint width;
@property (readonly) GLuint height;

@end


#pragma mark -AGLKTextureLoader
@interface AGLKTextureLoader : NSObject

+ (AGLKTextureInfo *)textureWithCGImage:(CGImageRef)cgImage
options:(NSDictionary *)options
   error:(NSError **)outError;

@end
```

AGLKTextureInfo是一个封装了纹理缓存的有用信息的简单类，例如相应的OpenGL ES纹理缓存的标识符以及纹理的图像尺寸。AGLKTextureLoader只声明了一个方法——“+textureWithCGImage:options:error:”。

AGLKTextureLoader的实现展现了Core Graphics和OpenGL ES的整合，提供了与GLKit的GLKTextureLoader相似的功能。在“+textureWithCGImage:options:error:”方

法中对于 OpenGL ES 函数的调用完成了标准的缓存管理步骤，包括生成、绑定和初始化一个新的纹理缓存，参见下面粗体显示的代码：

```
////////////////////////////////////////////////////////////////
// This method generates a new OpenGL ES texture buffer and
// initializes the buffer contents using pixel data from the
// specified Core Graphics image, cgImage. This method returns an
// immutable AGLKTextureInfo instance initialized with
// information about the newly generated texture buffer.
//    The generated texture buffer has power of 2 dimensions. The
// provided image data is scaled (re-sampled) by Core Graphics as
// necessary to fit within the generated texture buffer.
+ (AGLKTextureInfo *)textureWithCGImage:(CGImageRef)cgImage
   options:(NSDictionary *)options
   error:(NSError **)outError;
{
   // Get the bytes to be used when copying data into new texture
   // buffer
   size_t width;
   size_t height;
   NSData *imageData = AGLKDataWithResizedCGImageBytes(
      cgImage,
      &width,
      &height);

// Generation, bind, and copy data into a new texture buffer
GLuint      textureBufferID;

glGenTextures(1, &textureBufferID);                    // Step 1
glBindTexture(GL_TEXTURE_2D, textureBufferID);         // Step 2

glTexImage2D(                                          // Step 3
   GL_TEXTURE_2D,
   0,
   GL_RGBA,
   width,
   height,
   0,
   GL_RGBA,
   GL_UNSIGNED_BYTE,
   [imageData bytes]);

// Set parameters that control texture sampling for the bound
// texture
glTexParameteri(GL_TEXTURE_2D,
   GL_TEXTURE_MIN_FILTER,
   GL_LINEAR);
```

```
    // Allocate and initialize the AGLKTextureInfo instance to be
    // returned
    AGLKTextureInfo *result = [[AGLKTextureInfo alloc]
       initWithName:textureBufferID
       target:GL_TEXTURE_2D
       width:width
       height:height];

    return result;
}
```

glGenTextures() 和 glBindTexture() 函数与用于顶点缓存的命名方式相似的函数的工作方式相同。但是，glTexImage2D(GLenum target, Glint level, Glint internalFormat, GLsizei width, GLsizei height, Glint border, GLenum format, GLenum type, const GLvoid *data) 函数是 OpenGL ES 标准中最复杂的函数之一。它复制图片像素的颜色数据到绑定的纹理缓存中。glTextImage2D() 函数的第一个参数是用于 2D 纹理的 GL_TEXTURE_2D。第二个参数用于指定 MIP 贴图的初始细节级别。如果没有使用 MIP 贴图，第二个参数必须是 0。如果开启了 MIP 贴图，使用第二个参数来明确地初始化每个细节级别，但是要小心，因为从全分辨率到只有一纹素的每个级别都必须被指定，否则 GPU 将不会接受这个纹理缓存。

glTexImage2D() 的第三个参数是 internalFormat，用于指定在纹理缓存内每个纹素需要保存的信息的数量。对于 iOS 设备来说，纹素信息要么是 GL_RGB, 要么是 GL_RGBA。GL_RGB 为每个纹素保存红、绿、蓝三种颜色元素。GL_RGBA 保存一个额外的用于指定每个纹素透明度的透明度元素。

glTexImage2D() 函数的第四个和第五个参数用于指定图像的宽度和高度。高度和宽度需要是 2 的幂。border 参数一直是用来确定围绕纹理的纹素的一个边界的大小，但是在 OpenGL ES 中它总是被设置为 0。第七个参数 format 用于指定初始化缓存所使用的图像数据中的每个像素所要保存的信息，这个参数应该总是与 internalFormat 参数相同。其他的 OpenGL 版本可能在 format 和 internalFormat 参数不一致时自动执行图像数据格式的转换。

倒数第二个参数用于指定缓存中的纹素数据所使用的位编码类型，可以是下面的符号值之一：GL_UNSIGNED_BYTE、GL_UNSIGNED_SHORT_5_6_5、GL_UNSIGNED_SHORT_4_4_4_4 和 GL_UNSIGNED_SHORT_5_5_5_1。使用 GL_UNSIGNED_BYTE 会提供最佳色彩质量，但是它每个纹素中每个颜色元素的保存需要一字节的存储空间。结果是每次取样一个 RGB 类型的纹素，GPU 都必须最少读取 3 字节（24 位），每个 RGBA 类型的纹素需要读取 4 字节（32 位）。其他的纹素格式使用多种编码方式来把每个纹素的所有颜色元素的信息保存在 2 字节（16 位）中。GL_UNSIGNED_SHORT_5_6_5 格式把 5 位用于红色，6 位用于绿色，5 位用于蓝色，但是没有透明度部分。GL_UNSIGNED_SHORT_4_4_4_4 格式平均为每个纹素的颜色元素使用 4 位。GL_UNSIGNED_SHORT_5_5_5_1 格式为红、绿、蓝各使用 5 位，但透明度只使用 1 位。使

用 GL_UNSIGNED_SHORT_5_5_5_1 格式会让每个纹素要么完全透明，要么完全不透明。

不管为每个颜色元素保存的位数量是多少，颜色元素的强度最终都会被 GPU 缩放到 0.0 到 1.0 的范围内。一个强度为满值的颜色元素（所有那个颜色元素的位都是 1）对应于一个 1.0 的强度。透明度颜色元素强度为 1.0 表示完全不透明，透明度强度为 0.5 表示 50% 透明度，透明度强度为 0.0 则表示完全透明。

glTexImage2D() 的最后一个参数是一个要被复制到绑定的纹理缓存中的图片的像素颜色数据的指针。

“+textureWithCGImage:options:error:”方法使用 AGLKDataWithResizedCGImage-Bytes() 函数来获取用于初始化纹理缓存的内容的字节。AGLKDataWithResizedCGImageBytes() 是在 AGLKTextureLoader.m 中实现的，并且包含转换一个 Core Graphics 图像为 OpenGL ES 可用的合适字节的代码。

```
//////////////////////////////////////////////////////////////////
// This function returns an NSData object that contains bytes
// loaded from the specified Core Graphics image, cgImage. This
// function also returns (by reference) the power of 2 width and
// height to be used when initializing an OpenGL ES texture buffer
// with the bytes in the returned NSData instance. The widthPtr
// and heightPtr arguments must be valid pointers.
static NSData *AGLKDataWithResizedCGImageBytes(
   CGImageRef cgImage,
   size_t *widthPtr,
   size_t *heightPtr)
{
   NSCParameterAssert(NULL != cgImage);
   NSCParameterAssert(NULL != widthPtr);
   NSCParameterAssert(NULL != heightPtr);

   size_t originalWidth = CGImageGetWidth(cgImage);
   size_t originalHeight = CGImageGetWidth(cgImage);

   NSCAssert(0 < originalWidth, @"Invalid image width");
   NSCAssert(0 < originalHeight, @"Invalid image width");

   // Calculate the width and height of the new texture buffer
   // The new texture buffer will have power of 2 dimensions.
   size_t width = AGLKCalculatePowerOf2ForDimension(
      originalWidth);
   size_t height = AGLKCalculatePowerOf2ForDimension(
      originalHeight);

   // Allocate sufficient storage for RGBA pixel color data with
   // the power of 2 sizes specified
   NSMutableData    *imageData = [NSMutableData dataWithLength:
      height * width * 4];  // 4 bytes per RGBA pixel
```

```
NSCAssert(nil != imageData,
   @"Unable to allocate image storage");

// Create a Core Graphics context that draws into the
// allocated bytes
CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
CGContextRef cgContext = CGBitmapContextCreate(
   [imageData mutableBytes], width, height, 8,
   4 * width, colorSpace,
   kCGImageAlphaPremultipliedLast);
CGColorSpaceRelease(colorSpace);

// Flip the Core Graphics Y-axis for future drawing
CGContextTranslateCTM (cgContext, 0, height);
CGContextScaleCTM (cgContext, 1.0, -1.0);

// Draw the loaded image into the Core Graphics context
// resizing as necessary
CGContextDrawImage(cgContext, CGRectMake(0, 0, width, height),
   cgImage);

CGContextRelease(cgContext);

*widthPtr = width;
*heightPtr = height;
   return imageData;
}
```

Core Graphics 函数会把指定的cgImage拖入imgaeData提供的字节中。Core Graphics把cgImage拖入一个适当大小的Core Graphics上下文中，这个过程的一个副作用是把图像的尺寸调整为了2的幂。图像在被绘制的时候还被翻转了。翻转Y轴是必须的，因为Core Graphics是以原点在左上角同时Y轴向下增大的形式来实现iOS中的图片保存的。OpenGL ES的纹理坐标系会放置原点在左下角，同时Y值向上增大。翻转Y轴确保了图像字节拥有适用于纹理缓存的正确的方向。

注意 Mac OS X版Core Graphics保存图像的方式跟OpenGL ES一样都是以原点在左下角，同时Y值向上增大的方式来保存的。在建立iOS时，苹果修改了Core Graphics的实现，导致了Mac OS X上不存在的与OpenGL ES的轻微不兼容。

在cgImage被拖入imageData提供的字节之后，函数会返回imgaeData和数据对应的高度和宽度。

AGLKTextureLoader.m文件内剩下的代码是不言自明的。只有一个小细节的实现有点生疏，就是为用于初始化AGLKTextureInfo类的一个方法的实现和声明所使用的一个Objective-C类别（category）。下面代码块中粗体标注的就是类别语句。

```
//////////////////////////////////////////////////////////////////////////
// Instances of AGLKTextureInfo are immutable once initialized
@interface AGLKTextureInfo (AGLKTextureLoader)

- (id)initWithName:(GLuint)aName
   target:(GLenum)aTarget
   width:(size_t)aWidth
   height:(size_t)aHeight;

@end
```

Objective-C 类别会向现存的类添加方法，并使跨多个文件实现一个类成为可能。在这个例子中，类别只是为了使“-initWithName:target:width:height:”方法的声明与类的主接口相分离。在 AGLKTextureLoader.m 文件而不是 .h 文件中声明类别表示这个方法应该只用在 AGLKTextureLoader.m 文件中。这个类别的名字可以随意，但是命名它为 AGLKTextureLoader 进一步强调了添加的这个方法是要在 AGLKTextureLoader 类中使用的。

3.4 OpenGLES_Ch3_3 示例

摆弄一下例子 OpenGLES_Ch3_3 呈现的用户界面。它演示了各种效果，包括纹理的取样模式、纹理的循环模式，以及当一个纹理被映射到顶点并修改在视口中的位置时的图像失真。图 3-5 演示了几种效果。

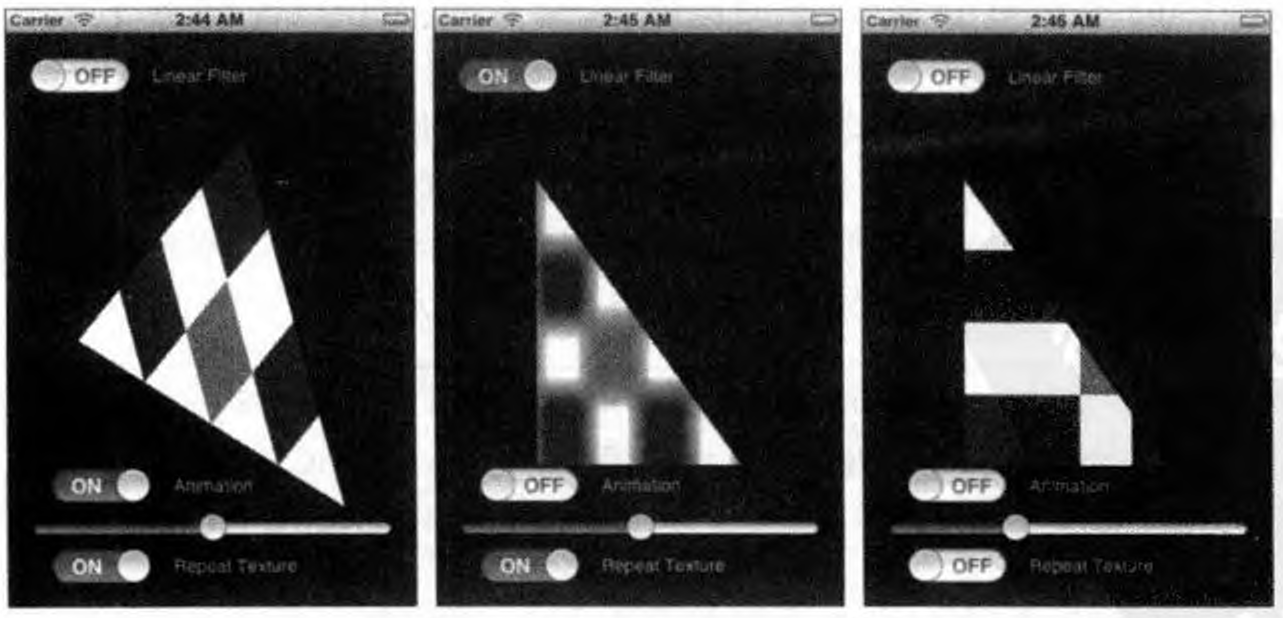


图 3-5 OpenGLES_Ch3_3 示例的截图显示了失真、过滤器和循环模式

图 3-5 中间的截图演示了一个 GL_LINEAR 过滤器放大一个小纹理来填充整个三角形的效果。另外两个截图显示的是 GL_NEAREST 过滤器的效果。

例子 OpenGLES_Ch3_3 向前一个例子创建的 AGLKVertexAttribArrayBuffer 类添加了一个“-reinitWithAttribStride:numberOfVertices:data:”方法。定期地用变化了的顶点位置重新初始化顶点数组缓存的内容以产生一个意在突出纹理失真的简单动画，这个失真是自然发生的几何纹理映射的一部分。

纹理取样模式和纹理循环模式被保存在 OpenGL ES 上下文的每个纹理的标识符中。换句话说，在上下文中的每个纹理缓存都有它自己相应的不依赖于任何其他的纹理缓存的模式。模式是通过 glTexParameteri() 函数来设置的，并且通常在一个纹理缓存被初始化以后就不会再更改它的模式了。但是，例子 OpenGLES_Ch3_3 修改了模式以响应用户的输入。这个例子使用下面的类别扩展了 GLKit 的 GLKEffectProptyTexture 类。

```
@implementation GLKEffectPropertyTexture (AGLKAdditions)

- (void)aglkSetParameter:(GLenum)parameterID
   value:(GLint)value;
{
   glBindTexture(self.target, self.name);

   glTexParameteri(
      self.target,
      parameterID,
      value);
}

@end
```

方法“-aglkSetParameter:value:”是使用 aglk 前缀命名的，这样避免了任何未来的方法命名冲突的可能性。如果添加的方法简单地被叫做“-setParameter:value:”，那么如果苹果有一天用相同的名字添加了一个方法到 GLKEffectPropertyTexture，这些方法就会发生冲突。Objective-C 在运行时最终会选择其中的一个方法，但是对于程序员来说可能就很难分辨出运行的到底是哪一个实现了。

3.5 透明度、混合和多重纹理

OpenGL ES 提供了如此多的选项用来保存、映射，以及配置纹理，以至于这个主题常常会压垮程序员。关于纹素颜色格式、纹理环境函数、取样模式，以及细节级别存在一个天文数字的组合和排列。本节甚至会介绍更多的选项，因此为了保持这个主题在可管理范围内，专注于现代最常用的用例是必要的。

可以使用包含透明度元素的 GL_RGBA 纹理格式来指定每个纹素的透明度。图 3-6 显示了一个拥有透明度颜色元素值的纹理，因此棋盘格图案可以透过半透明的纹素显示出来。通常一个或者更多个纹素会结合灯光和顶点颜色来决定每个片元的最终颜色和透

明度。第 4 章会介绍灯光。每个片元产生的透明度都会影响片元怎么与一个帧缓存内的现存内容相混合。

图 3-6 含有透明度颜色元素的纹理会在半透明纹素的位置显示图案

当纹理计算出来一个完全不透明的最终片元颜色时，这个片元颜色会简单地替换任何在帧缓存的像素颜色渲染缓存内现存的对应的像素颜色。还有更有意思的混合方式。如果计算出来的片元颜色部分透明或者全透明，OpenGL ES 会使用一个混合函数来混合片元颜色与像素颜色渲染缓存内对应的像素。

通过调用 glEnable(GL_BLEND) 函数来开启混合。然后通过调用 glBlendFunc(sourceFactor, destinationFactor) 来设置混合函数。sourceFactor 参数用于指定每个片元的最终颜色元素是怎么影响混合的。destinationFactor 参数用于指定在目标帧缓存中已经存在的颜色元素会怎么影响混合。最常用的混合函数配置是设置 sourceFactor 为 GL_SRC_ALPHA，设置 destinationFactor 为 GL_ONE_MINUS_SRC_ALPHA，如下代码所示：

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

GL_SRC_ALPHA 用于让源片元的透明度元素挨个与其他的片元颜色元素相乘。GL_ONE_MINUS_SRC_ALPHA 用于让源片元的透明度元素（1.0）与在帧缓存内的正被更新的像素的颜色元素相乘。结果是，如果片元的透明度值为 0，那么没有片元的颜色会出现在帧缓存中。如果片元的透明度值为 1，那么片元的颜色会完全替代在帧缓存中的对应的像素颜色。介于 0.0 到 1.0 之间的透明度值意味着片元颜色的一部分会被添加到帧缓存内对应的像素颜色的一部分中来产生一个混合的结果。当使用 glBlendFunc(GL_SRC_ALPHA，GL_ONE_MINUS_SRC_ALPHA) 时，在帧缓存中的最终颜色是用下面的方程式计算的：

$$Red_{final} = Alpha_{fragment} * Red_{fragment} + (1.0 - Alpha_{fragment}) * Red_{Frame\ Buffer}$$
$$Green_{final} = Alpha_{fragment} * Green_{fragment} + (1.0 - Alpha_{fragment}) * Green_{Frame\ Buffer}$$
$$Blue_{final} = Alpha_{fragment} * Blue_{fragment} + (1.0 - Alpha_{fragment}) * Blue_{Frame\ Buffer}$$
$$Alpha_{final} = Alpha_{fragment} + (1.0 - Alpha_{fragment}) * Alpha_{Frame\ Buffer}$$

3.5.1 在 OpenGLES_Ch3_4 示例中混合片元颜色

使用 glBlendFunc(GL_SRC_ALPHA，GL_ONE_MINUS_SRC_ALPHA) 会与 iOS Core Graphics 的“正常混合模式”产生相同的结果。图 3-7 是例子 OpenGLES_Ch3_4 产生的显示结果。图 3-7 中的树叶纹理与在帧缓存的像素颜色渲染缓存中的黑色像素混合，

因此在树叶纹理包含透明纹素的每个地方黑色像素都保持不变。然后使用一个虫子图片的第二个纹理会与像素颜色渲染缓存相混合。每当虫子纹理的纹素是透明的时候，在帧缓存内的已存的颜色就保持不变。最终的渲染结果为虫子纹理在树叶纹理之上，树叶纹理在黑色背景之上。

注意 在图3-7中显示的层效果揭示了iOS Core Animation技术的基本原理。每个Core Animation层使用一个对应的OpenGL ES的像素颜色渲染缓存来保存像素颜色数据。每个层的像素颜色数据作为一个OpenGL ES纹理缓存，并且纹理缓存会使用glBlendFunc(GL_SRC_ALPHA，GL_ONE_MINUS_SRC_ALPHA)函数来与后帧缓存混合在一起。

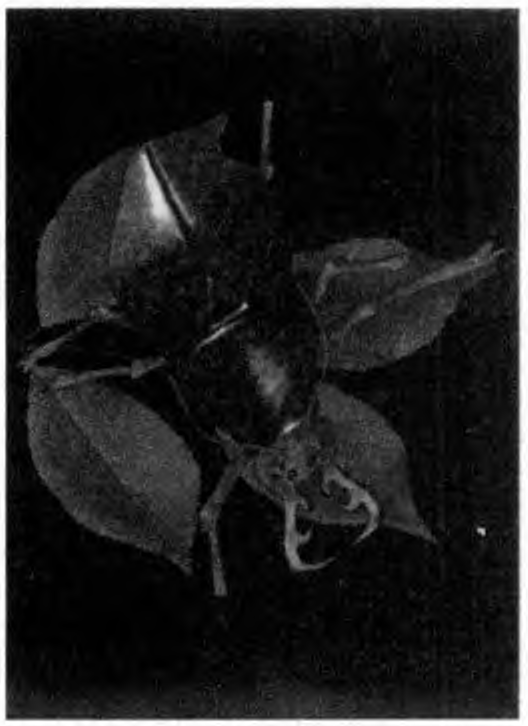


图3-7 一个虫子纹理在树叶纹理之上，树叶纹理在黑色背景之上

例子OpenGLES_Ch3_4在例子OpenGLES_Ch3_3中的“-viewDidLoad”方法的实现中添加了下面的粗体代码。该代码会加载第二个纹理并开启与像素颜色渲染缓存的混合。

```
// Setup texture0
CGImageRef imageRef0 =
   [[UIImage imageNamed:@"leaves.gif"] CGImage];

self.textureInfo0 = [GLKTextureLoader
   textureWithCGImage:imageRef0
   options:[NSDictionary dictionaryWithObjectsAndKeys:
      [NSNumber numberWithBool:YES],
      GLKTextureLoaderOriginBottomLeft, nil]
   error:NULL];

// Setup texture1
CGImageRef imageRef1 =
   [[UIImage imageNamed:@"beetle.png"] CGImage];

self.textureInfo1 = [GLKTextureLoader
   textureWithCGImage:imageRef1
   options:[NSDictionary dictionaryWithObjectsAndKeys:
      [NSNumber numberWithBool:YES],
      GLKTextureLoaderOriginBottomLeft, nil]
   error:NULL];

// Enable fragment blending with Frame Buffer contents
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

相比上一个例子的另一个小变化是在从图像加载纹理时应用了一个 NSDictionary 对象来设定选项。在这个例子中，GLKTextureLoaderOriginBottomLeft 键与布尔值 YES 搭配是为了命令 GLKit 的 GLKTextureLoader 类垂直翻转图像数据。这个翻转可以抵消图像的原点与 OpenGL ES 标准原点之间的差异。

```
[NSDictionary dictionaryWithObjectsAndKeys:
   [NSNumber numberWithBool:YES],
    GLKTextureLoaderOriginBottomLeft, nil]
```

例子 OpenGLES_Ch3_4 把相同的几何图形渲染了两次：第一次使用了一个纹理，第二次使用了另一个。混合发生在每次被一个纹理着色的一个片元与在像素颜色渲染缓存中已存的像素颜色混合的时候。下面来自 OpenGLES_Ch3_4 的“-glkView:drawInRect:”实现的摘录显示了相关的代码：

```
self.baseEffect.texture2d0.name = self.textureInfo0.name;
self.baseEffect.texture2d0.target = self.textureInfo0.target;
[self.baseEffect prepareToDraw];

// Draw triangles using the first three vertices in the
// currently bound vertex buffer
[self.vertexBuffer drawArrayWithMode:GL_TRIANGLES
   startVertexIndex:0
   numberOfVertices:6];

self.baseEffect.texture2d0.name = self.textureInfo1.name;
self.baseEffect.texture2d0.target = self.textureInfo1.target;
[self.baseEffect prepareToDraw];

// Draw triangles using the first three vertices in the
// currently bound vertex buffer
[self.vertexBuffer drawArrayWithMode:GL_TRIANGLES
   startVertexIndex:0
   numberOfVertices:6];
```

GLKit 的 baseEffect 是由第一个纹理设定的，同时 vertexBuffer 被绘制。然后 baseEffect 由第二个纹理设定，同时 vertexBuffer 被再次绘制。这两个过程中也伴随着与像素颜色渲染缓存的混合。绘图的顺序决定了哪一个纹理会出现在另一个之上。在当前情况下是虫子在树叶的上面。纹理绘制的顺序颠倒过来的话会把树叶置于虫子之上。

3.5.2 示例 OpenGLES_Ch3_5 中的多重纹理

很多有用的可视效果可以通过把片元颜色与在像素颜色渲染缓存中现存的颜色相混合来实现，但是这个技术有两个主要的缺点：每次显示更新时几何图形必须要被渲染一到更多次，混合函数需要从像素颜色渲染缓存读取颜色数据以便与片元颜色混合。然后

结果被写回帧缓存。当带有透明度数据的多个纹理如图 3-7 所示层叠时，每个纹理的像素颜色渲染缓存的颜色会被再次读取、混合、重写。如例子 OpenGLES_Ch3_4 所示通过多次读写像素颜色渲染缓存来创建一个最终的渲染像素的过程叫做多通道渲染。如往常一样，内存访问限制了性能，因此多通道渲染是次优的。接下来将介绍的多重纹理方法避免了多通道渲染的大部分缺陷。

所有的现代GPU都能够同时从至少两个纹理缓存中取样纹素。GLKit的GLKBaseEffect类同时支持两种纹理。执行纹素取样和混合的硬件组件叫做一个纹理单元或者一个取样器。如果你的应用需要超过两个纹理单元，在确定一个单独的通道中可以结合多少个纹理之前，请使用下面的代码：

```
GLint iUnits;
glGetIntegerv(GL_MAX_TEXTURE_UNITS, &iUnits);
```

例子 OpenGLES_Ch3_5 产生的输出与 OpenGLES_Ch3_4 相同，除了树叶和虫子纹理被混合在了一个通道中，而且与像素颜色渲染缓存的内容混合来产生结果片元颜色的过程只会在每次显示更新中发生一次。

多重纹理引入了另一个组合组配置选项。为了帮助降低复杂性，iOS 5 中的 GLKit 的GLKEffectPropertyTexture 类操作了 3 种常见的多重纹理模式：GLKTextureEnvModeReplace、GLKTextureEnvMode Modulate，以及 GLKTextureEnvModeDecal。GLKEffectPropertyTexture默认使用GLKTextureEnvModeModulate模式，这种模式几乎总是产生最好的结果。GLKTextureEnvModeModulate 模式会让所有的为灯光和其他效果计算出来的颜色与从一个纹理取样的颜色相混合。

GLKEffectPropertyTexture 的 envMode 属性用于配置混合模式。OpenGLES_Ch3_5 与 OpenGES_Ch3_4 加载相同的两个纹理，但是不再需要明确地启动与帧缓存的像素颜色渲染缓存的混合。相反，baseEffext 的第二个纹理属性 texture2D1 被设置为使用 GLKTextureEnvModeDecal 模式，这种模式会使用一个与 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) 类似的方程式来混合第二个与第一个纹理。

```
:self.baseEffect.texture2d1.envMode = GLKTextureEnvModeDecal;
```

需要为所有的纹理单元的使用提供纹理坐标。多重纹理支持为每个纹理使用不同的纹理坐标。在本书例子中的 SceneVertex 数据类型可以被扩展来保存每个顶点的纹理坐标的多重集合，但是 OpenGLES_Ch3_5 配置 OpenGL ES 2.0 为两个纹理使用相同的纹理坐标。下面的来自例子 OpenGLES_Ch3_5 的“-glkView:drawInRect:”方法的完整实现显示了具体步骤。粗体的代码突出了每个纹理单元的纹理坐标的设定：

```
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect
{
   // Clear back frame buffer (erase previous drawing)
```

```
[(AGLKContext *)view.context clear:GL_COLOR_BUFFER_BIT];

[self.vertexBuffer prepareToDrawWithAttrib:GLKVertexAttribPosition
   numberOfCoordinates:3
   attribOffset:offsetof(SceneVertex, positionCoords)
   shouldEnable:YES];

[self.vertexBuffer prepareToDrawWithAttrib:GLKVertexAttribTexCoord0
   numberOfCoordinates:2
   attribOffset:offsetof(SceneVertex, textureCoords)
   shouldEnable:YES];

[self.vertexBuffer prepareToDrawWithAttrib:GLKVertexAttribTexCoord1
   numberOfCoordinates:2
   attribOffset:offsetof(SceneVertex, textureCoords)
   shouldEnable:YES];

[self.baseEffect prepareToDraw];

// Draw triangles using the first three vertices in the
// currently bound vertex buffer
[self.vertexBuffer drawArrayWithMode:GL_TRIANGLES
   startVertexIndex:0
   numberOfVertices:6];
}
```

3.5.3 在 OpenGLES_Ch3_6 示例中自定义纹理

多重纹理的强大和灵活性在使用自定义的利用 OpenGL ES Shading Language 的 OpenGL ES 2.0 片元程序时会变得更加明显。一个额外的例子，在 OpenGLES_Ch3_6 中，首先用一个由 GLKit 的 GLKBaseEffect 在后台自动生成的 Shading Language 程序绘制一个立方体，然后使用下面的自定义顶点和片元的 Shading Language 程序绘制第二个立方体。现在不用担心没有学过 Shading Language，但是如果你很好奇，可以摆弄一下 OpenGLES_Ch3_6，看看它是怎么工作的。

```
//
//  Shader.vsh
//  TestShaders
//
//

attribute vec4 aPosition;
attribute vec3 aNormal;
attribute vec2 aTextureCoord0;
attribute vec2 aTextureCoord1;

varying lowp vec4 vColor;
```

```
varying lowp vec2 vTextureCoord0;
varying lowp vec2 vTextureCoord1;

uniform mat4 uModelViewProjectionMatrix;
uniform mat3 uNormalMatrix;

void main()
{
    vec3 eyeNormal = normalize(uNormalMatrix * aNormal);
    vec3 lightPosition = vec3(0.0, 0.0, 1.0);
    vec4 diffuseColor = vec4(0.7, 0.7, 0.7, 1.0);

    float nDotVP = max(0.0, dot(eyeNormal, normalize(lightPosition)));

    vColor = vec4((diffuseColor * nDotVP).xyz, diffuseColor.a);
    vTextureCoord0 = aTextureCoord0.st;
    vTextureCoord1 = aTextureCoord1.st;

    gl_Position = uModelViewProjectionMatrix * aPosition;
}
```

一个片元着色器是一个由 GPU 执行的，用来完成计算当前渲染缓存中的每个片元的最终颜色所需要的运算的简短程序。包含片元着色器程序的文件通常使用“.fsh”文件扩展名。

```
//
//  Shader.fsh
//  TestShaders
//

uniform sampler2D uSampler0;
uniform sampler2D uSampler1;
varying lowp vec4 vColor;
varying lowp vec2 vTextureCoord0;
varying lowp vec2 vTextureCoord1;

void main()
{
   // first texture Modulate, second texture Decal
   lowp vec4 color0 = texture2D(uSampler0, vTextureCoord0);
   lowp vec4 color1 = texture2D(uSampler1, vTextureCoord1);
   gl_FragColor = mix(color0, color1, color1.a) * vColor;
}
```

相比其他的纹理混合配置方法，GL Shading Language 程序往往是既简短又更加自文档化的。但是，OpenGL ES Shading Language 是一个复杂到值得用整本书来论述的主题，例如 OpenGL Shading Language（第 3 版），由 Randi J. Rost、Bill Licea-

Kane、Dan Ginsburg、John M. Kessenich、Barthold Lichtenbelt、Hugh Malan 以及 Mike Weiblen 所著。本书会逐步地讲解一些概念同时在几个章节中做例子演示，但是本书不是一本完整的参考书。

3.6 纹理压缩

苹果建议使用 GPU 所支持的纹理压缩方式。纹理压缩最小化了当要保存相对高细节级别的纹理时保存纹理缓存所需要的内存的数量。截止到 iOS 5，所有的 iOS 设备都支持 PowerVR 纹理压缩（PVRTC）格式。但是，这无法保证将来的苹果硬件还会支持 PVRTC 格式。因此，在使用它之前，程序有必要检查非标准 OpenGL ES 的 GL_IMG_texture_compression_pvrtc 扩展的存在性。苹果建议使用下面的 C 函数来检查这个扩展：

```
BOOL CheckForExtension(NSString *searchName)
{
   // For best results, extensionsNames should be stored so that it
   // does not need to be recreated on each invocation.
   NSString *extensionsString = [NSString
      stringWithCString:glGetString(GL_EXTENSIONS) encoding:
      NSASCIIStringEncoding];
   NSArray *extensionsNames = [extensionsString
      componentsSeparatedByString:@" "];

   return [extensionsNames containsObject:searchName];
}
```

一个程序可以使用下面形式的代码来检查是否支持纹理压缩。

```
if( CheckForExtension(@"GL_IMG_texture_compression_pvrtc") )
{
   // The extension is available.
}
```

作为 iOS SDK 的一部分，苹果提供了命令行实用工具 texturetool。texturetool 工具会将“.png”图像转换为能够直接加载到压缩的纹理缓存的格式，这个网址：http://developer.apple.com/iphone/library/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TextureTool/ TextureTool.html 做了介绍。GLKit 的 GLKTextureLoader 类也支持压缩的纹理。

用来初始化压缩纹理的图像的尺寸需要是 2 的幂，同时它们还需要是正方形。一个 256×256 像素的图像可以用在一个压缩纹理缓存中，但是一个 64×128 像素的图像却不能。

苹果为使用压缩纹理提供了示例代码，地址为：http://developer.apple.com/iphone/

library/samplecode/PVRTextureLoader/。

3.7 小结

本章几乎提供了实现一个 2D 硬件加速的基于 sprite 的游戏所需的全部信息。一个 sprite 仅仅是一个可以移动的包含透明像素的图像，例子 OpenGLES_Ch3_3 甚至演示了在显示器上来回移动纹理的一个方法，同时间接演示了由纹理取样产生的失真效果。苹果在 http://developer.apple.com/iphone/library/samplecode/GLImageProcessing 网址提供的示例代码中显示了如何用多重纹理来实现强大的图像处理效果，例如对图像进行亮度、对比度、饱和度、色度和锐度调整。这些例子提示了商业图像处理应用之下的苹果硬件加速 Core Image 技术的实现。

截止到现在，所有的例子程序都是使用 3D 技术绘制简单的 2D 三角形。第 2 章介绍了在一个 iOS 设备上显示任何基于 OpenGL ES 的图像所需的所有步骤。第 3 章扩展了第 2 章的例子实现在三角形上显示多重纹理。第 4 章会在整个三维上深入探讨渲染。灯光是在一个二维屏幕上显示出三维假象的关键。

第 4 章　散发一些光线

3D 计算机图形技术会把物体的数学描述转换成逼真的幻象。人类大脑可以解释在平面屏幕上显示的图片，还可以部分基于几何图形的暗示感知不存在的深度。例如，感觉小的物体要比大的物体更远一些。如果一个物体盖住另一个物体的一部分，人脑就可以感觉到前后。但是，要呈现有深度的错觉就需要有模拟的灯光效果。观察一下在图 4-1 中相同的几何数据的两个渲染结果。左边的图片在渲染时没有使用灯光，右边使用了模拟灯光的图片会更加逼真。

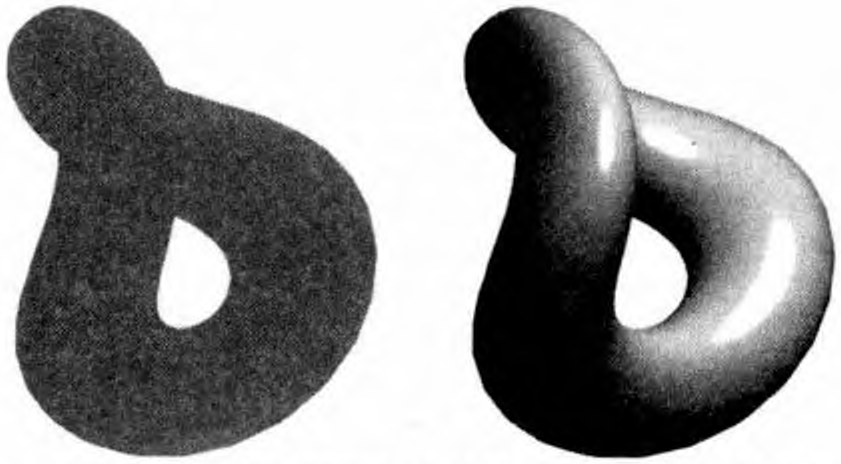


图 4-1　在使用和不使用模拟灯光的情况下渲染相同的几何数据

本章会介绍用于模拟 3D 物体与光线之间的相互作用的多个技术。OpenGL ES 使用 GPU 来计算一个场景中的几何图形投射和散发出的模拟光线的数量。通常，GPU 首先为每个三角形的每个顶点执行光线计算，然后把计算的结果插补在顶点之间来修改每个渲染的片元的最终颜色。因此模拟灯光的质量和光滑度要取决于组成每个 3D 物体的顶点的数量。模拟灯光的另一种方法会增强或者替代传统的模拟方法。在许多情况下，灯光效果可以被预计算并被“烘焙”进纹理中，以便可以在完全不需要 GPU 光线计算的情况下产生逼真的场景。然而另一种方法会利用 GPU 的能力为渲染场景中的每个单独的片元分别计算和应用灯光效果。

本章会介绍灯光模拟背后的概念，利用 GLKit 并使用相对简单的应用代码演示灯光效果。GLKit 复制了 OpenGL ES 1.x 的传统灯光能力，同时为大部分应用提供了适当

的内建的灯光模拟方式。在介绍完灯光的基础之后，本章会描述一些 OpenGL ES 2.0 提供的更高级方法。

4.1　环境光、漫反射光、镜面反射光

OpenGL ES 灯光模拟由每个光源的三个截然不同的部分组成：环境光、漫反射光、镜面反射光。程序会分别配置每个部分的颜色。图 4-2 显示的是模拟灯光的每个部分的效果。环境光来自各个方向，因此会同等地增强所有几何图形的亮度。程序通过设置模拟环境光的颜色和亮度来设置场景中的背景灯光的基础水平。环境光的颜色会着色所有的几何图形，因此一个红色的环境光会让一个场景中的所有几何图形对象显现红色或者粉红色。每个光源的漫反射部分是定向的，会基于三角形相对于光线的方向来照亮场景中的每个三角形。如果一个三角形的平面垂直于光线的方向，那么漫反射光会直接投射到三角形上，并被剧烈地散射开来，这会让三角形显得灯光通明。如果三角形的平面是平行于光线的方向或者背离光线的方向，那么几乎没有任何光线会照射到三角形上，因此漫反射光很少或者根本不会对三角形的亮度产生影响。漫反射光的颜色只会着色被定向的光线照射到的三角形。最后从几何图形对象反射出来的光线叫做镜面反射光。镜面物体会反射大量的光线，但是钝面的物体不会。因此镜面反射光的感知亮度是由照射到每个三角形上的光线的量和三角形的反光度决定的。镜面反射部分的颜色决定了闪光点的颜色。

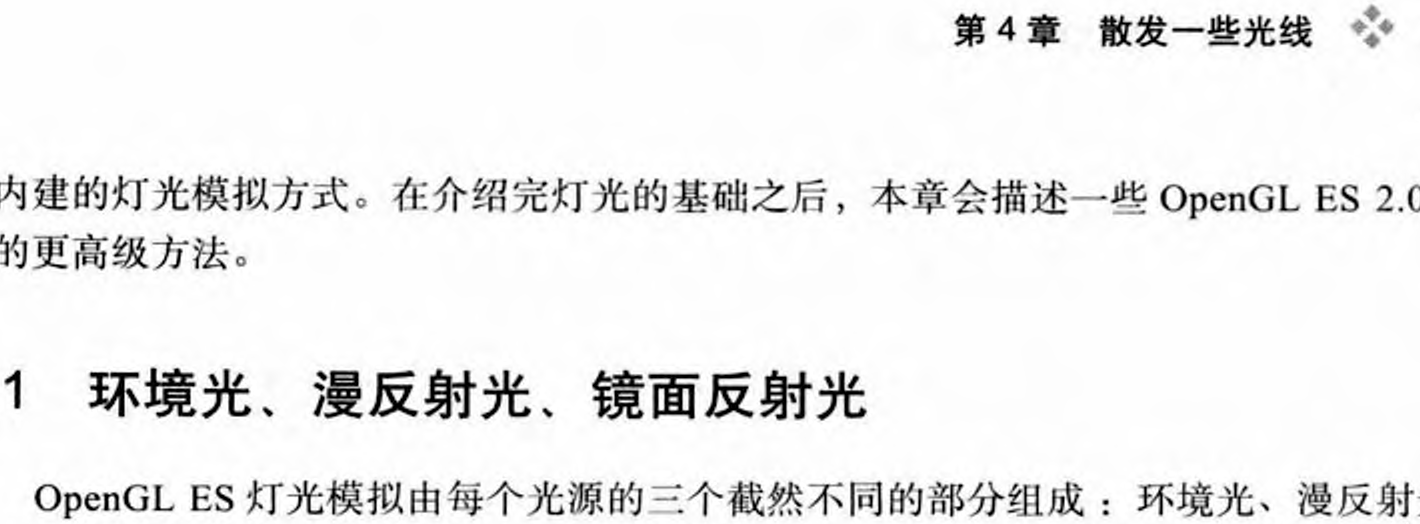


图 4-2　对于模拟的环境光、漫反射光、镜面反射光的感知依赖于被光线照射的每个三角形的平面

图 4-2 描述了射向一个平面的光线。箭头指示了光线的方向。图片还表明每个平面有两个面：一个面向光源的前面和一个背离光源的后面。

一个渲染的三角形中的每个光线组成部分的效果取决于三个相互关联的因素：光线的设置、三角形相对于光线的方向，以及三角形的材质属性。OpenGL ES 使用术语材质来描述三角形的一个属性，这个属性会影响三角形与光线的各个组成部分之间的相互作

用。反射或散射的光线越多，三角形看起来就会越灯光通明。高反射率的镜面反射光会让三角形显现类似镀铬或者镜面的光泽。低镜面反射率会呈现一个类似于尘土和地毯的钝的材料的错觉。

只有每个光源的环境光部分才会照射三角形的后面。一些 OpenGL ES 实现支持双面灯光的选项，就是三角形的前面和后面分别有不同的材质，但是 iOS 5 不支持这个选项。

注意 材质对于呈现高度逼真的场景是非常重要的。人类无法直接看到一个光源发出的光，除非看向光源。我们可以感知到周围的物体是因为一定量的光线经过环境中的每一个对象的反射后进入了人眼。因此，我们通常看到的光是已经与材质相互作用后的光。但是，就像现实中的物体很少是由单一颜色的三角面组成的一样，现实物体很少只有一个材质，而且真实的材质往往要比 OpenGL ES 灯光模拟能够复制的材质要复杂得多。OpenGL ES 灯光模拟仅仅靠自己是不足以渲染出一个带有磨损痕迹的光泽球或者一个泛着油光的水泥路面的。尤其需要知道的是材质是由 OpenGL ES 支持以用在一个程序需要利用逼真的动态灯光绘制一个闪亮的水晶花瓶时，但是在绝大多数的程序中应该将材质设置为默认值，并使用纹理来产生想要的效果。

OpenGLES_Ch4_1 示例程序演示了怎样使用苹果的 iOS 5 中的 GLKit 来开启和配置 OpenGL ES 2.0 灯光。灯光是 OpenGL ES 1.x 与 OpenGL ES 2.0 完全不同的一个地方。早期的版本至少支持 8 个并存的灯光。相比普通应用以前使用的，在 OpenGL ES 1.x 中存在更多的选项和配置可能。OpenGL ES 2.0 根本不会直接地支持任何的灯光。作为替代，运行在 GPU 上的自定义 Shading Language 程序可以自由地以任何适合应用的方式来实现光线计算。GPU 能够快速地执行 4.2 节介绍的灯光运算，但是灯光模拟仍然会增加计算机消耗。GLKit 的 GLKBaseEffect 类会自动地生成用于复制 OpenGL ES 1.x 为至少三个灯光所做的光线计算的 Shading Language 程序。如果可能的话，只使用一个光源或者不使用光源是比较好的。在第 4.5 节会解释使用最少的消耗来渲染出最逼真的灯光效果的最常用的技术。

4.2 计算有多少光线照向每个三角形

在渲染时无论是使用 OpenGL ES 1.x 灯光模拟还是备选方案，光线与 3D 几何图形的相互作用都起了重要作用。光线与几何图形相互作用的关键是要计算出每个几何物体照射和发散出来多少光线。要做到这一点，就需要计算出每个三角形有多么接近于与光的方向相垂直的方向。线性代数提供了解决方案。三角形可以由三个顶点来定义。第 1 章介绍了怎么计算任意两个顶点之间的矢量。因此使用一个顶点和两个矢量来定义一个三

角形也是可行的，参见图 4-3。

如在第 1 章提到的，很多方便的运算都可以用矢量来完成。最重要的矢量运算之一是矢量积：提供两个矢量，这两个矢量的矢量积定义了第三个垂直于前两个矢量的矢量，参见图 4-4。

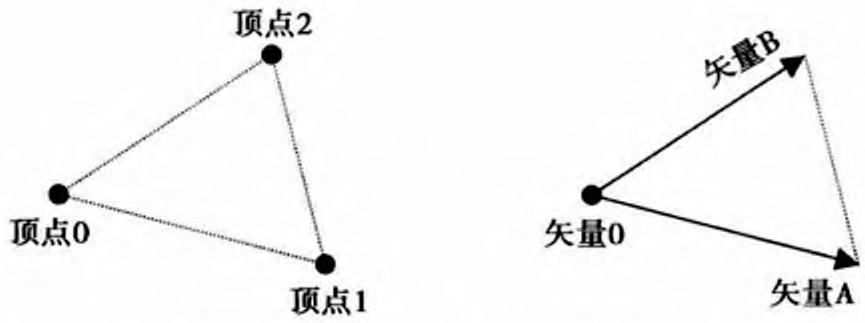


图 4-3　使用两种不同的方式来定义相同的三角形

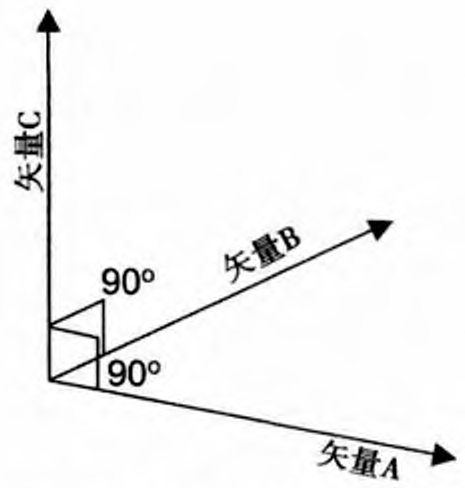


图 4-4　矢量 C 是矢量 A 和矢量 B 的矢量积

下面的方程式使用减法和乘法代替三角函数来计算两个矢量的矢量积；GPU 可以快速地执行减法和乘法运算。

```
VectorC.x = VectorA.y * VectorB.z – VectorA.z * VectorB.y;
VectorC.y = VectorA.z * VectorB.x – VectorA.x * VectorB.z;
VectorC.z = VectorA.x * VectorB.y – VectorA.y * VectorB.x;
```

注意　VectorB × VectorA 的矢量积和 VectorA × VectorB 矢量积的方向是相反的。一个常见的错误是无意中计算出了相反的矢量积，这会使光线扭曲并使每个三角形的前面变成后面。记住矢量顺序的最简单的方法是“右手原则”，在网页：http://en.wikipedia.org/wiki/Right-hand_rule 上做了很好的解释。如果你正俯视一个三角形，三角形的顶点是以逆时针顺序计数的。如图 4-4 所示，从所有矢量交接的顶点开始，以逆时针方向的下一个顶点是矢量 A 末尾的那个顶点，最后一个顶点在矢量 B 的末尾。

光线计算依赖于表面法向量（surface normal vector），或者简称法向量（normal vector）。可以为任何一个三角形计算出一个法向量：法向量的方向垂直于一个三角形的平面并且法向量可以使用定义三角形的任意两个矢量的矢量积计算出来。法向量也是单位向量，这意味着一个法向量的大小（也称为长度）总是 1.0。

任何矢量都可以转换成一个单位向量，通过用这个矢量的长度除以这个矢量的每个分量。结果是一个与原先的矢量方向相同的并且长度等于 1.0 的新矢量。因此，为了计算一个法向量，首先需要计算矢量积向量，然后用这个矢量积向量的长度除以矢量积的

每个分量。这个操作是如此的常见以至于转换矢量为单位矢量通常被称为“标准化”操作。例子程序 OpenGLES_Ch4_1 使用下面的代码来计算法向量：

```
//////////////////////////////////////////////////////////////////////////
// Returns a unit vector in the same direction as the cross
// product of vectorA and VectorB
GLKVector3 SceneVector3UnitNormal(
   const GLKVector3 vectorA,
   const GLKVector3 vectorB)
{
   return GLKVector3Normalize(
      GLKVector3CrossProduct(vectorA, vectorB));
}
```

矢量积是用 GLKit 的 GLKVector3.h 头文件内的一个内联函数来计算的。下面的实现匹配实际的 GLKit 实现并且已经为显示做了格式化：

```
//////////////////////////////////////////////////////////////////////////
// Returns the Cross Product vectorA x vectorB
GLKVector3 GLKVector3CrossProduct(
   GLKVector3 vectorA,
   GLKVector3 vectorB)
{
   GLKVector3 result = {
      vectorA.y * vectorB.z - vectorA.z * vectorB.y,
      vectorA.z * vectorB.x - vectorA.x * vectorB.z,
      vectorA.x * vectorB.y - vectorA.y * vectorB.x
   };

   return result;
}
```

GLKit 的 GLKVector3Normalize（GLKVector3 vectorA）函数需要多点解释。这个函数的目的是返回一个与矢量 A 方向相同但是大小等于 1.0 的单位向量。一个矢量的大小是这个矢量的长度，这个长度可以用标准距离公式$\sqrt{\text{vectorA. x2} + \text{vectorA. y2} + \text{vectorA. z2}}$计算得到。

下面的对于 SceneVector3Length() 和 SceneVector3Normalize() 函数的实现与对应的 GLKVector3Length() 和 GLKVector3Normalize() 函数的实现类似。在产品代码中应该使用 GLKit 函数以受益于苹果在将来的框架版本中提供的维护和性能的增强。SceneVector3 实现为显示格式化了代码并显示了算术错误检查：

```
//////////////////////////////////////////////////////////////////////////
// Returns the length a.k.a. magnitude of vectorA
GLfloat SceneVector3Length(
   const GLKVector3 vectorA)
{
   GLfloat length = 0.0f;
```

```
    GLfloat lengthSquared =
       (vectorA.x * vectorA.x) +
       (vectorA.y * vectorA.y) +
       (vectorA.z * vectorA.z);


    if(FLT_EPSILON < lengthSquared)
    {  // avoid square root of zero error if lengthSquared
       // is too small
       length = sqrtf(lengthSquared);
    }

    return length;
}
```

在前面的 SceneVector3Length() 函数中使用的常量 FLT_FPSILON 是在 iOS SDK 提供的标准 float.h 头文件中定义的。FLT_EPSILON 常量是一个非常小的在浮点数学计算的过程中不会被舍入为 0 的正数。重要的是要确保 lengthSquared 不会太接近于 0 以防止不经意间计算了 0 的平方根。

在知道了一个矢量的长度以后，又实现了 SceneVector3Normalize(GLKVector3 vectorA) 函数来缩放矢量 A 为一个单位向量，具体代码如下：

```
//////////////////////////////////////////////////////////////////////////////
// Returns a Unit Vector with the same direction as vectorA
GLKVector3 SceneVector3Normalize(
   GLKVector3 vectorA)
{
   const GLfloat  length = SceneVector3Length(vectorA);
   float          oneOverLength = 0.0f;

   if(FLT_EPSILON < length)
   {  // avoid divide by zero if length too small
      oneOverLength = 1.0f / length;
   }

   GLKVector3 result = {
      vectorA.x * oneOverLength;
      vectorA.y * oneOverLength;
      vectorA.z * oneOverLength;
   };

   return result;
}
```

投射到三角形上的光线的数量可以通过确定光线的方向与法向量的方向之间的角度轻松计算出来。再次，线性代数提供了一个快速方便的解决方案。两个单位矢量之间的标量积可以计算出介于两个矢量之间的角的余弦，参见图 4-5。介于一个光线方向上的

单位向量和一个三角形的表面法向量之间的角度的余弦决定了照射到三角形上的光线的数量。如果三角形垂直于光线，那么表面法向量会平行于光线的方向，因此它们之间的角度就是 0 度。0.0 度的余弦值是 1.0，这意味着最大强度的光会照射到三角形上。如果这个三角形平行于光线的方向，那么介于光线和表面法向量之间的角度就是 90.0 度或者 -90.0 度（余弦值都是 0.0），因此没有光线会照射到三角形上。0.0 度和 90.0 度之间的角度的余弦值在 0.0 至 1.0 之间，它决定了照射到三角形上的光线的数量。

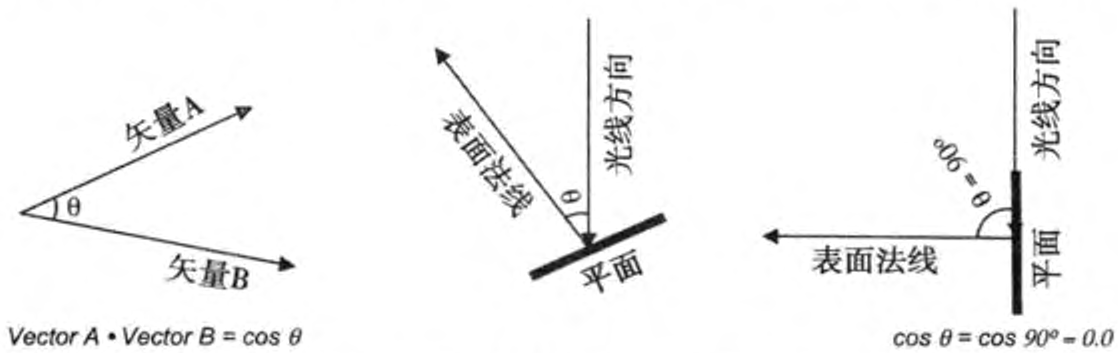


图 4-5　单位矢量 A 和 B 之间的标量积等于 cos θ

OpenGL ES 程序为每个顶点指定了单独的法向量。法向量通常计算一次，然后与顶点的位置和纹理坐标一起被保存起来。标准 OpenGL ES 1.1 灯光模拟会使用下面的方程式用 GPU 为每个顶点计算光线的方向与法向量之间的标量积。

```
DotProduct = (VectorA.x * VectorB.x) + (VectorA.y * VectorB.y) +
(VectorA.z * VectorB.z);
```

如果一个三角形的三个顶点被赋予了相同的法向量，这叫做平面法线，灯光模拟会让三角形显得平坦。如果每个顶点的法向量是包含顶点的所有三角形的法向量的一个平均值，灯光模拟会创建三角形被轻微弯曲的错觉，如图 4-6 所示。

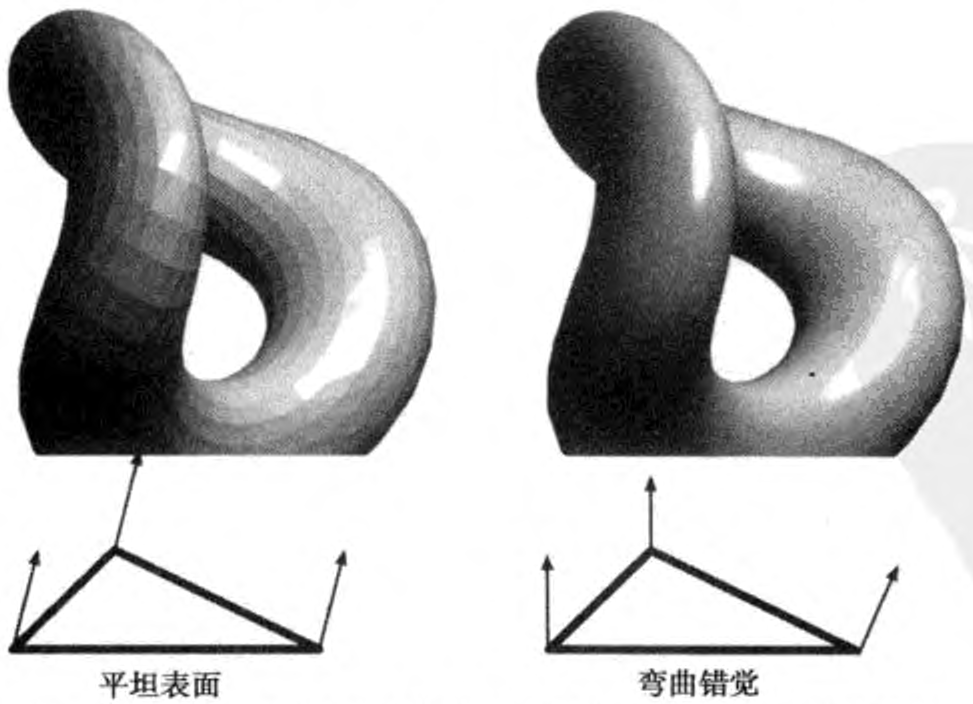


图 4-6　同一几何图形的两个渲染结果显示了法向量的效果

4.3　使用 GLKit 灯光

iOS 5 的 GLKit 使用与下面摘录的用于漫反射光的代码相似的 Shading Language 代码为每个光线实现了标准的灯光模拟方程式：

```
vec3 eyeNormal = normalize(uNormalMatrix * aNormal);
float nDotVP = max(0.0, dot(eyeNormal, ulightDirection));
vColor = vec4((uDiffuseColor * nDotVP).xyz, uDiffuseColor.a);
```

现在不用担心不理解 Shading Language 摘录。这里提供它只是为了显示出 GPU 实现灯光模拟的方式的特点。这个代码会改变每个顶点的法向量来匹配正在被渲染的场景的方向，计算光线方向与新的法向量之间的标量积，然后使用这个标量积来按比例决定光的漫反射颜色的影响力。GLKit 会自动地生成与摘录相似的 Shading Language 代码，以便同时混入每个光线的镜面发射光和环境光颜色。

GLKit 的 GLKBaseEffect 类最多支持三个命名为 light0、light1 和 light2 的模拟灯光。每个灯光都由一个相同名字的 GLKBaseEffect 属性所代表。这些属性是 GLKEffectPropertyLight 类的实例，这些属性反过来又包含用于设置每个灯光的属性。每个灯光至少有一个位置、一个环境颜色、一个漫反射颜色和一个镜面反射颜色。每个灯光都可以被单独地开启和关闭。下面的代码显示了怎么使用 GLKit 灯光。粗体代码配置了 light0。

```
// Create a base effect that provides standard OpenGL ES 2.0
// Shading Language programs and set constants to be used for
// all subsequent rendering
self.baseEffect = [[GLKBaseEffect alloc] init];
self.baseEffect.light0.enabled = GL_TRUE;
self.baseEffect.light0.diffuseColor = GLKVector4Make(
   0.7f, // Red
   0.7f, // Green
   0.7f, // Blue
   1.0f);// Alpha
self.baseEffect.light0.position = GLKVector4Make(
   1.0f,
   1.0f,
   0.5f,
   0.0f);
```

灯光的漫反射颜色被设置为不透明中等灰色。灯光的镜面反射和环境颜色保持为 GLKit 的默认值，分别为不透明白色和不透明黑色。这意味着灯光的漫反射部分不会影响场景并且高反光的物体会显得非常有光泽。

这段代码使用一个带有四个元素的 GLKVector4 来设置光源的位置。前三个元素要么是光源的 X、Y 和 Z 位置，要么是指向一个无限远的光源的方向。第四个元素指定

了前三个元素是一个位置还是一个方向。如果第四个元素是零，前三个元素就是一个方向；如果第四个元素非零，光源会从它的位置向各个方向投射光线。因此每个顶点的光线方向是多种多样的，并且必须要由 GPU 使用从每个顶点到光源的方向而计算出来。位置光源可以被设置为以一个锥形而不是向各个方向发散光线的聚光灯。6.3 节包含了位置聚光灯的示例代码，参考资料可以在网址：http://developer.apple.com/mac/library/documentation/Darwin/Reference/ManPages/man3/glLight.3.html 上找到。

仅仅开启和配置一个灯光是不够的，必须为被一个灯光照射的每一个三角形的每一个顶点提供法向量。4.4 节的示例 OpenGLES_Ch4_1 会按需重新计算法向量，但是大部分应用是预先计算法向量的或者从一个数据文件一起加载法向量和顶点位置信息。

在一个或多个 GLKBaseEffect 的灯光被开启后，灯光决定了渲染的物体的颜色；GLKBaseEffect 的常量颜色和所有的顶点颜色被忽略了。例子 OpenGLES_Ch4_1 使用一个开启的灯光绘制了场景，然后使用 GLKBaseEffect 的 constantColor 属性（而不是使用一个灯光）绘制了线段。需要两个独立的 GLKBaseEffect 实例，因为在 iOS 5 中，在一个 GLKBaseEffect 的灯光被创建以后，constantColor 属性就被忽略了，即使 GLKBaseEffect 的灯光被关闭了。例如，“baseEffect.light0.enabled = GL_FALSE;”。

注意 作为最后的手段，GLKBaseEffect 的 constantColor 属性会告诉 GLKBaseEffect 为生成的片元使用什么颜色。大部分 OpenGL ES 应用使用某种灯光和纹理的结合来决定片元的颜色。constantColor 属性仅适用于渲染单调不发光的物体。

4.4 OpenGLES_Ch4_1 示例

例子 OpenGLES_Ch4_1 演示了基于 GLKit 的灯光模拟，并为浏览法向量的效果提供了多个选项。图 4-7 显示了 OpenGLES_Ch4_1 应用的屏幕截图。一个简单的钻出平面的四面角锥体。从后面照向角锥体的右边的定向光会让角锥体的某些面比其他的面更亮。使用平面法线矢量会产生一个多面的外观。平均法向量会让四面角锥体看起来就像是一个光滑的圆锥体。图 4-7 中的第三个屏幕截图使用渲染的线描绘了计算出来的法向量和光线的方向。

移动 OpenGLES_Ch4_1 应用的滑块来升高和降低角锥体的顶点，这会改变角锥体的面与光线方向之间的角度。每次顶点的位置发生改变，法向量就会被重新计算。

计算法向量使用了两个不同的方式。当使用平面法线矢量时，每个三角形的每个顶点拥有相同的法向量，因此三角形看起来是平的。否则，每个顶点的法向量就是共享顶点的所有三角形的法向量的一个平均值。这个例子提供了一个用于切换是否开启法向量绘制的用户接口和一个指示光线方向的线。在移动滑块以显示由于法向量与光线方向之

间的角度变化所产生的明暗变化时，请仔细检查法向量。

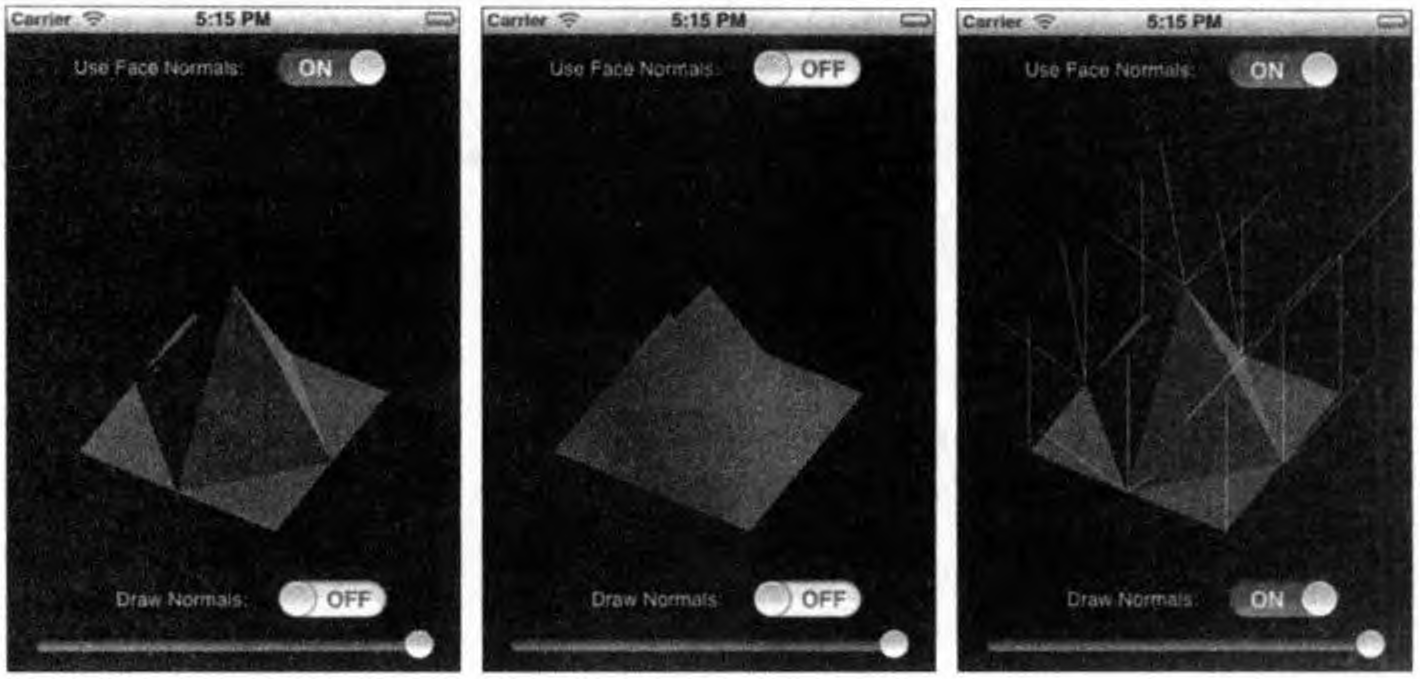


图 4-7　OpenGLES_Ch4_1 例子程序的屏幕截图

图 4-8 展示了在例子 OpenGLES_Ch4_1 中使用的几何图形。从 A 开始到 I 所标识的九个顶点定义了标识为 0 到 7 的 8 个三角形。用户界面中的滑块会改变顶点 E 的 Z 坐标，顶点 E 定义了三角锥的高度。当顶点 E 上下移动时，三角形 2 到 5 会改变。图 4-8 等同于图 4-7 中显示的相同的三角形。例子 OpenGLES_Ch4_1 是使用侧俯视角来渲染三角形的，参见图 4-7。图 4-8 是使用顶部直视视角来描述三角形的。

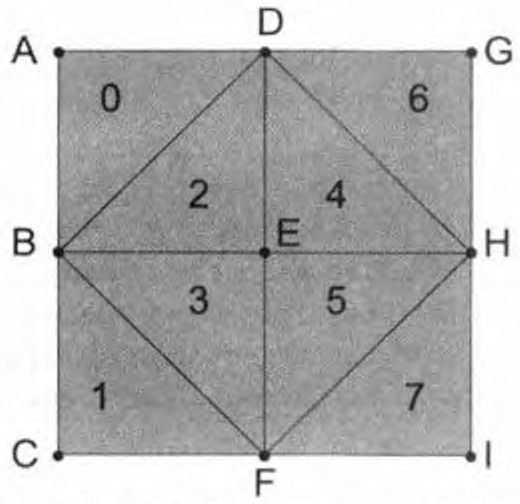


图 4-8　例子 OpenGLES_Ch4_1 中的顶点和三角形之间的关系

在 OpenGLES_Ch4_1ViewController.m 中使用下面的代码定义和初始化变量和 vertexA 到 vertexI。

```
//////////////////////////////////////////////////////////////////
// Define the positions and normal vectors of each vertex in the
// example.
SceneVertex vertexA = {{-0.5,  0.5, -0.5}, {0.0, 0.0, 1.0}};
SceneVertex vertexB = {{-0.5,  0.0, -0.5}, {0.0, 0.0, 1.0}};
SceneVertex vertexC = {{-0.5, -0.5, -0.5}, {0.0, 0.0, 1.0}};
SceneVertex vertexD = {{ 0.0,  0.5, -0.5}, {0.0, 0.0, 1.0}};
SceneVertex vertexE = {{ 0.0,  0.0,  0.0}, {0.0, 0.0, 1.0}};
SceneVertex vertexF = {{ 0.0, -0.5, -0.5}, {0.0, 0.0, 1.0}};
SceneVertex vertexG = {{ 0.5,  0.5, -0.5}, {0.0, 0.0, 1.0}};
SceneVertex vertexH = {{ 0.5,  0.0, -0.5}, {0.0, 0.0, 1.0}};
SceneVertex vertexI = {{ 0.5, -0.5, -0.5}, {0.0, 0.0, 1.0}};
```

新数据类型 SceneTriangle 声明了一个用于保存定义一个三角形的三个顶点的结构体。本例使用 OpenGLES_Ch4_1ViewController 类的实例的一个数组来保存在图 4-8 中描述的 8 个三角形。

```
//////////////////////////////////////////////////////////////////
// This data type is used to store information for triangles
typedef struct {
   SceneVertex vertices[3];
}
SceneTriangle;


//////////////////////////////////////////////////////////////////
@interface OpenGLES_Ch4_1ViewController ()
{
   SceneTriangle triangles[8];
}

@end
```

如下代码所示，OpenGLES_Ch4_1ViewController 的“-viewDidLoad”方法创建了叫做 baseEffect 和 extraEffect 的 GLKBaseEffect 实例。粗体代码设置了 baseEffect 的灯光之一。

```
// Create a base effect that provides standard OpenGL ES 2.0
// shading language programs and set constants to be used for
// all subsequent rendering
self.baseEffect = [[GLKBaseEffect alloc] init];
self.baseEffect.light0.enabled = GL_TRUE;
self.baseEffect.light0.diffuseColor = GLKVector4Make(
   0.7f, // Red
   0.7f, // Green
   0.7f, // Blue
   1.0f);// Alpha
self.baseEffect.light0.position = GLKVector4Make(
   1.0f,
   1.0f,
   0.5f,
   0.0f);

extraEffect = [[GLKBaseEffect alloc] init];
self.extraEffect.useConstantColor = GL_TRUE;
self.extraEffect.constantColor = GLKVector4Make(
   0.0f, // Red
   1.0f, // Green
   0.0f, // Blue
   1.0f);// Alpha
```

在“-viewDidLoad”中的下一段代码块提供了在第 5 章将要讲解的技术的一个预览。在 OpenGLES_Ch4_1 例子中的整个场景被旋转并定位以更容易地看到三角锥的高度变化。注释掉或者删掉下面的代码块，然后重建并运行 OpenGLES_Ch4_1 工程以从上向下观察三角锥。

```
{  // Comment out this block to render the scene top down
   GLKMatrix4 modelViewMatrix = GLKMatrix4MakeRotation(
      GLKMathDegreesToRadians(-60.0f), 1.0f, 0.0f, 0.0f);
   modelViewMatrix = GLKMatrix4Rotate(
      modelViewMatrix,
      GLKMathDegreesToRadians(-30.0f), 0.0f, 0.0f, 1.0f);
   modelViewMatrix = GLKMatrix4Translate(
      modelViewMatrix,
      0.0f, 0.0f, 0.25f);

   self.baseEffect.transform.modelviewMatrix = modelViewMatrix;
   self.extraEffect.transform.modelviewMatrix = modelViewMatrix;
}
```

在图 4-8 中的从 0 到 7 的 8 个三角形被使用如下的从 vertexA 到 vertexI 的顶点初始化，然后三角形被存储在一个顶点属性数组缓存中以供 GPU 使用。

```
triangles[0] = SceneTriangleMake(vertexA, vertexB, vertexD);
triangles[1] = SceneTriangleMake(vertexB, vertexC, vertexF);
triangles[2] = SceneTriangleMake(vertexD, vertexB, vertexE);
triangles[3] = SceneTriangleMake(vertexE, vertexB, vertexF);
triangles[4] = SceneTriangleMake(vertexD, vertexE, vertexH);
triangles[5] = SceneTriangleMake(vertexE, vertexF, vertexH);
triangles[6] = SceneTriangleMake(vertexG, vertexD, vertexH);
triangles[7] = SceneTriangleMake(vertexH, vertexF, vertexI);
// Create vertex buffer containing vertices to draw
self.vertexBuffer = [[AGLKVertexAttribArrayBuffer alloc]
   initWithAttribStride:sizeof(SceneVertex)
   numberOfVertices:sizeof(triangles) / sizeof(SceneVertex)
   data:triangles
   usage:GL_DYNAMIC_DRAW];
```

下一个代码块显示了 OpenGLES_Ch4_1ViewController 的“-glkView:drawInRect:”方法的整个实现。为绘制准备好 GLKit baseEffect。像素颜色渲染缓存被清除。每个顶点的位置和法线属性被准备好以供 GPU 使用。粗体代码发送每个顶点的法向量给 GPU。使用 baseEffect 渲染的三角形包含由 GLKit 在后台自动生成的 Shanding Language 程序提供的模拟灯光。最后，如果 OpenGLES_Ch4_1ViewController 的 shouldDrawNormals 属性值是 YES，那么 OpenGLES_Ch4_1ViewController 的“-drawNormals”方法会被调用。

```
//////////////////////////////////////////////////////////////////
// GLKView delegate method: Called by the view controller's view
// whenever Cocoa Touch asks the view controller's view to
// draw itself. (In this case, render into a frame buffer that
// shares memory with a Core Animation Layer)
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect
{
   [self.baseEffect prepareToDraw];

   // Clear back frame buffer (erase previous drawing)
   [(AGLKContext *)view.context clear:GL_COLOR_BUFFER_BIT];

   [self.vertexBuffer prepareToDrawWithAttrib:GLKVertexAttribPosition
      numberOfCoordinates:3
      attribOffset:offsetof(SceneVertex, position)
      shouldEnable:YES];
   [self.vertexBuffer prepareToDrawWithAttrib:GLKVertexAttribNormal
      numberOfCoordinates:3
      attribOffset:offsetof(SceneVertex, normal)
      shouldEnable:YES];

   // Draw triangles using vertices in the currently bound vertex
   // buffer
   [self.vertexBuffer drawArrayWithMode:GL_TRIANGLES
      startVertexIndex:0
      numberOfVertices:sizeof(triangles) / sizeof(SceneVertex)];

   if(self.shouldDrawNormals)
   {
      [self drawNormals];
   }
}
```

shouldDrawNormals 属性同 centerVertexHeight 和 shouldUseFaceNormals 属性一起在 OpenGLES_Ch4_1ViewController.h 中声明。更新属性值的代码在 OpenGLES_Ch4_1ViewController 类的实现中。每当一个如滑块或者切换器的用户界面对象改变时，OpenGLES_Ch4_1ViewController 会使用 Cocoa Touch Target-Action 设计模式。Target-Action 设计模式的帮助文档网址为：http://developer.apple.com/iphone/library/documentation/General/ Conceptual/Devpedia-CocoaApp/TargetAction.html。本质上接收一个单独的对象参数的方法会被用户界面对象调用来更新应用的状态。改变状态的特定对象是用户界面对象的目标（target）。这个方法叫做动作（action）。按照惯例，动作方法的参数是调用这个动作的用户界面对象。

很多动作方法使用下面的形式来声明：

```
- (IBAction)takeShouldUseFaceNormalsFrom:(UISwitch *)sender;
- (IBAction)takeShouldDrawNormalsFrom:(UISwitch *)sender;
- (IBAction)takeCenterVertexHeightFrom:(UISlider *)sender;
```

Xcode 的“.storyboard”编辑器使可视化用户界面构建成为可能，并包含了目标和动作的详细说明。在例子 OpenGLES_Ch4_1 中，每当用户移动了滑块，一个滑块用户界面就会调用“-takeCenterVertexHeightFrom:”动作。

“-takeCenterVertexHeightFrom:”的实现是非常直白的，如下代码所示。

```
////////////////////////////////////////////////////////////////
// This method sets the value of the center vertex height to the
// value obtained from sender
- (IBAction)takeCenterVertexHeightFrom:(UISlider *)sender;
{
   self.centerVertexHeight = sender.value;
}
```

当 centerVertexHeight 属性值变化时，包含图 4-8 中的 vertexE 的三角形也会变化。当三角形变化时，三角形的法向量必须被重新计算。这个重新计算是例子 OpenGLES_Ch4_1 的关键。

OpenGLES_Ch4_1ViewController 通过实现属性的自定义访问器方法来拦截属性的改变。访问器是在第 2 章中介绍的，它们是被特别命名的方法，可以被调用来修改属性值。当编译类似“self.centerVertexHeight = sender.value;”的点标记语法时，Objective-C 编译器会自动生成对于访问器方法的调用。用来设置 centerVertexHeight 属性的值的方法被命名为“-setCenterVertexHeight:”。

```
////////////////////////////////////////////////////////////////
// This method sets the value of centerVertexHeight and updates
// vertex normals
- (void)setCenterVertexHeight:(GLfloat)aValue
{
   centerVertexHeight = aValue;

   SceneVertex newVertexE = vertexE;
   newVertexE.position.z = self.centerVertexHeight;

   triangles[2] = SceneTriangleMake(vertexD, vertexB, newVertexE);
   triangles[3] = SceneTriangleMake(newVertexE, vertexB, vertexF);
   triangles[4] = SceneTriangleMake(vertexD, newVertexE, vertexH);
   triangles[5] = SceneTriangleMake(newVertexE, vertexF, vertexH);

   [self updateNormals];
}
```

每次“-setCenterVertexHeight:”方法被调用来改变 centerVertexHeight 属性的值时，

vertexE 的高度（Z 分量）也改变了，并且包含 vertexE 的四个三角形也被重新创建。之后 OpenGLES_Ch4_1ViewController 的“-updateNormals”方法被调用来重新计算受影响后的法向量。

```
//////////////////////////////////////////////////////////////////
// Called recalculate the normal vectors for the receiver's
// triangles using either face normals or averaged vertex normals.
- (void)updateNormals
{
   if(self.shouldUseFaceNormals)
   {  // Use face normal vectors to produce facets effect
      // Lighting Step 3
      SceneTrianglesUpdateFaceNormals(triangles);
   }
   else
   {  // Interpolate normal vectors for smooth rounded effect
      // Lighting Step 3
      SceneTrianglesUpdateVertexNormals(triangles);
   }

   // Reinitialize the vertex buffer containing vertices to draw
   [self.vertexBuffer
      reinitWithAttribStride:sizeof(SceneVertex)
      numberOfVertices:sizeof(triangles) / sizeof(SceneVertex)
      data:triangles];
}
```

包含法向量的三角形的顶点值被更新在顶点属性数组缓存中，以便在下一次场景被渲染时，它们可以被 GPU 使用。如果你对于法向量重新计算的数学计算过程比较好奇，请看一下在 OpenGLES_Ch4_1ViewController.m 文件中的 SceneTrianglesUpdateFaceNormals() 函数。它使用了在 4.2 节介绍过的 SceneVector3UnitNormal() 函数。

使用例子 OpenGLES_Ch4_1 做测试来影响渲染结果。试着改变灯光的位置。这个例子会为光源保持默认颜色值，试着为环境和镜面反射光部分指定颜色。试着添加第二个光源。

4.5 把灯光烘焙进纹理中

前面每个照射到的顶点，除了位置坐标，法向量的使用还需要三个浮点值（12 字节）的存储空间。当定义一个场景中的几何图形的顶点有几十万或上百万个时，每个顶点的额外的 12 字节会快速增加。额外的 12 字节不仅会消耗稀缺的 GPU 所控制的内存，而且每次几何图形被渲染时，它们还必须被 GPU 读取。

如果纹理可以以任何方式来使用，那么 OpenGL ES 灯光模拟的一个替代方法就是把灯光烘焙进一个现存的纹理中。换句话说，就是使用已经包含了本来是由灯光模拟生

成的明亮和黑暗区域的纹理。例子 OpenGLES_Ch4_2 会使用一个包含阴影灯光效果的纹理。

把灯光烘焙进纹理仅仅适用于几何图形和灯光都不是动态的情况下。如果灯光来回移动或者改变颜色，烘焙进的灯光看起来就不合适了。如果几何图形发生了较大的变化，以至于原先的黑色区域暴露在了光线之下，此时的纹理看起来就是错误的。第 6 章会讲解如何在不改变几何图形的情况下创建变化的场景。

如在 OpenGLES_Ch4_2 例子和图 4-9 中显示的一样，烘焙灯光效果可以呈现出一个更精细的错觉。

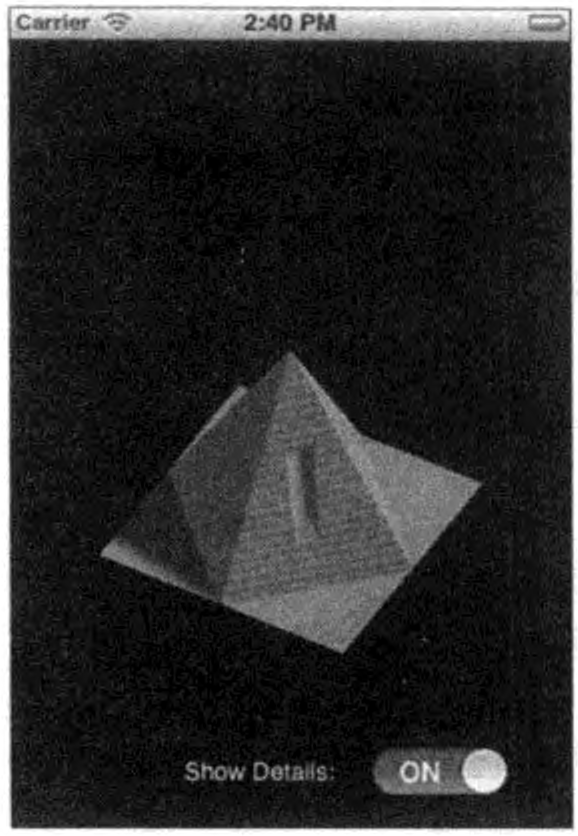


图 4-9　一个纹理所提供的精细的灯光和阴影的错觉

4.6　片元计算

如之前提到的，OpenGL ES 1.x 灯光模拟是 GLKit 通过计算光线方向矢量和每个顶点的顶点法向量之间的标量积来测定的。标量积决定了有多少光线会照向顶点。之后，计算出来的光线效果被插补在顶点之间使光线看起来平滑。

纹理为如图 4-9 所示的精细灯光的固定错觉控制渲染的每个片元的颜色。但是，在决定片元颜色时，GPU 已经执行了复杂的运算。是否可以使用 GPU 为每个单独的片元而不仅仅是为顶点重新计算灯光效果？当然可以，GPU 是可以通过编程来做到这一点的。

用于顶点的相同的方程式可以用在每个片元上。光线方程式需要每个片元的法向量。解决方法是编码一个纹理的每个 RGB 纹素内的法向量的 X、Y、Z 分量。这样的纹理称为法线贴图（normal map）。用于每个片元灯光的技术通常叫做法线贴图、凹凸贴图，或者 DOT3 灯光，因为这三个术语本质上都是描述的效果。

一个纹理单元决定了哪一个法线贴图纹素会影响片元。一个 Shading Language 程序会计算由选中的纹素和光线的方向所代表的矢量的标量积。然后使用这个标量积来按比例确定最终的片元的颜色效果。

每片元光线计算即使是在动态的灯光和几何图形的情况下，仍然可以工作良好。但是，每片元光线计算既需要嵌入式 GPU 做高耗费计算，又需要使用会耗尽 GPU 控制内存的法线贴图。生成法线贴图时也可能会出现问题。在一些情况下，iPhone SDK 附带的 PVRTextureTool 能够帮助生成法线贴图，但要让一切看起来刚刚好也是复杂的。由于早期工具支持、计算耗费和纹理的内存限制等原因，可用设备对于每片元光线计算的支持并不好。随着嵌入式 GPU 和工具支持的必然改进，高级每片元光线计算效果会变

得更实际。

在网址 http://en.wikipedia.org/wiki/Normal_mapping 中可以找到多个与每片元光线计算相关的优秀源码的更多信息。由 Aaftab Munshi、Dan Ginsburg 和 Dave Shreiner 所著的《OpenGL ES 2.0 Programming Guide》一书包含一个叫做“使用法线贴图的每像素光线计算”章节。Lighthouse3D 是一个专注于 3D 计算机图形的网站，并且是深入学习灯光主题的最好资源之一。网址 http://www.lighthouse3d.com/tutorials/glsl-tutorial/lighting/ 为实现标准的每片元灯光方程式提供了一个优秀的手把手教程。

4.7 小结

真实世界的灯光效果是我们感知周围环境的关键，模拟灯光效果可以为 3D 渲染增加真实感。结合了材质的传统 OpenGL ES 顶点灯光模拟提供了广泛的配置选项，但还要忍受各种限制以及嵌入式 GPU 的相对低的性能。比起 OpenGL ES 材质，纹理几乎总是会产生更高质量的渲染结果。在使用了纹理之后，可以在纹理中包含一些灯光效果，同时完全避免 OpenGL ES 灯光模拟。所有运行 iOS 的硬件都支持每片元灯光计算技术，并且随着内存、处理器和工具支持的改进这项技术会变得更加普遍。

截止到现在，所有渲染例子产生的都是类似一个裱在墙上的静止相片的冻结场景。下一章会讲解怎么从任意视点渲染一个场景，以及围着对象飞以从任意角度观察它们的知识。

第 5 章　改变你的视点

本章会讲解和演示从任意视点渲染几何对象的技术。3D 对象有正面、侧面、顶部和底部。变换（transformation）是一种相对简单的数学运算，它决定了在场景中的对象的哪一部分是可见的。很多人从心理上觉得可视化渲染时发生的变换效果是很难的。即使是变换效果已被准确地预计，数学计算和最终结果之间的关系可能也不是很明显。本章会使用例子来测试变换，然后提供常见的变换技巧。

5.1　深度渲染缓存（Depth Render Buffer）

三角形、线段和点是按它们被 GPU 处理的顺序被渲染的。如果没有一个深度渲染缓存（或者直接简称为深度缓存），为最后一个对象的绘制而产生的片元总是会覆盖以前渲染的层叠的片元。至今为止本书中的例子一直小心地控制绘图的顺序，以确保在没有深度缓存的情况下仍然可以产生正确的最终图像。

变换使从任意视点渲染场景成为可能。视点决定了场景中的哪一个对象会被渲染在其他对象的前面。例如，当从一个看向面部的视点渲染一个人类头部的几何图形时，所有组成头的后部的三角形都不会出现在渲染的场景中。然而，当依赖于渲染的顺序时，如果 GPU 最后处理的是头后部的三角形，那么即使不应该可见，最终它们也将是可见的。

在一些情况下，一个程序可以按照正确的渲染顺序从后向前排序对象。可以排序组成人类头部的三角形，以便组成面部的三角形是最后一个被 GPU 处理的并且覆盖组成头后部的三角形。不幸的是，排序三角形经常会导致重写顶点缓存的内容，并会击溃缓存带来的内存访问优化。如果视点变成面向头的一侧，那么组成面部的部分三角形和组成头后部的部分三角形在渲染场景中将都是可见的，但这需要一个不同的排序顺序。一个变化的视点必然需要重新计算所绘制对象的正确顺序。此外，光排序可能还不够。在一些情况下，三角形贯穿彼此并且正确地渲染应该包含两个三角形的片元，但是排序会强制一个三角形的所有片元覆盖另一个的片元。

深度渲染缓存是一个可选的输出缓存，并且与像素颜色渲染缓存相似。第 2 章为了

与像素颜色渲染缓存一起使用而介绍了“其他缓存”的概念，参见图 2-2。几乎所有的 OpenGL ES 应用都使用深度缓存，因为几乎所有的 OpenGL ES 应用都使用坐标系变换来改变渲染的视点。在大部分情况下，一个深度缓存会消除对于三角形、线段和点进行排序的需求。

注意 深度缓存常常也叫做 Z 缓存，因为如果坐标系的 X 轴和 Y 轴对应于屏幕的宽和高，那么 Z 轴指示的就是屏幕的内外。一个片元和视点之间的距离大体相当于这个片元沿着 Z 轴深入屏幕的位置。

每次渲染一个片元时，片元的深度（片元与视点之间的距离）被计算出来并与在深度缓存中为那个片元位置保存的值进行对比。如果这个片元的深度值更小（更接近视点），那么就用这个片元来替换在像素颜色渲染缓存中的那个片元位置的任何颜色，并用刚刚渲染的片元的深度值来更新深度缓存。如果一个片元的深度值比在深度缓存中保存的值更大，这意味着某些已经渲染的片元更接近于视点。在这种情况下，新的片元在还没有更新像素颜色渲染缓存的情况下就会被丢弃。

GPU 把对于每个片元的深度的计算作为渲染的一个固有部分。深度缓存的使用为 GPU 提供了一个用来保存计算出的深度的地方，之后这个深度又被 GPU 利用来控制在像素颜色渲染缓存中的片元的置换。

利用 GLKit 添加深度缓存

GLKView 类让添加深度缓存变得很容易。只要设置视图的 drawableDepthFormat 属性为 GLKViewDrawableDepthFormat16 或者 GLKViewDrawableDepthFormat24 而不是默认值即可，具体执行代码为“GLKViewDrawableDepthFormatNone:view.drawable-DepthFormat = GLKViewDrawableDepthFormat16；”。

GLKit 支持使用 16 位或者 24 位来保存深度值的深度渲染缓存。使用 16 位只可以表现 65 536 个不同的深度。如果两个片元的深度非常接近，那么深度缓存可能就没有足够的精度来区分它们。像素颜色渲染缓存中的最终片元颜色可能来自接近同一个深度的片元之一。事实上，这个结果有时也被称为深度冲突（Z-fighting），因为最终的片元颜色常常会在可能性之间来回闪烁并在渲染场景中制造一个可见的干扰。

使用 24 位来保存深度值可以区分 1700 万个不同的深度值，同时这是以消耗更多的 GPU 稀缺内存为代价的。即使是使用 24 位，当共面三角形重叠的时候还可能会产生深度冲突，但这不是很常见。

配置 OpenGL ES 状态的可选步骤允许程序改变 GPU 执行深度测试时所使用的函数。例如，调用 OpenGL ES glDepthFunc(GL_ALWAYS) 函数实际上会禁止深度测试，因为所有渲染的片元会替换像素颜色缓存中这个片元的位置的先前的任何颜色。默认的深度测试函数是 GL_LESS，它表示每个片元的颜色只有在该片元的深度值小于深度

缓存中该片元的位置所保存的值时才会替换像素颜色缓存的内容。更小的深度值意味着片元更接近视点。glDepthFunc() 指定的深度测试函数的完整集合是 GL_NEVER、GL_ALWAYS、GL_LESS、GL_LEQUAL、GL_EQUAL、GL_GEQUAL、GL_GREATER 或者 GL_NOTEQUAL。

在前面的章节中，glClear() 函数已经被用在例子中，它清除了缓存的内容。例如，调用 glClear(GL_COLOR_BUFFER_BIT) 会设置当前帧缓存的像素颜色缓存中的每个像素的值为 glClearColor() 函数设定的颜色。调用 glClear(GL_DEPTH_BUFFER_BIT) 会设置当前帧缓存的深度缓存中的每个值为最大深度值。两个缓存通常用一行代码来清除，通过使用 C 语言的位 OR 操作符将参数结合到 glClear() 中，例如 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)。

注意　几乎所有本书的例子所使用的 AGLKContext 中都包含“-setClearColor:”和“–clear:”方法，这两个方法封装了 OpenGL ES 的 glClearColor() 和 glClear() 函数。

5.2　例子 OpenGLES_Ch5_1 和例子 OpenGLES_Ch5_2

例子 OpenGLES_Ch5_1 和 OpenGLES_Ch5_2 都渲染一个代表地球的球体。定向灯光模拟从太阳照向地球的光线。一个映射到球体的纹理显示了熟悉的大陆、海洋和天气。图 5-1 的左图为来自 OpenGLES_Ch5_1 的屏幕截图，右图为来自 OpenGLES_Ch5_2 的截图。

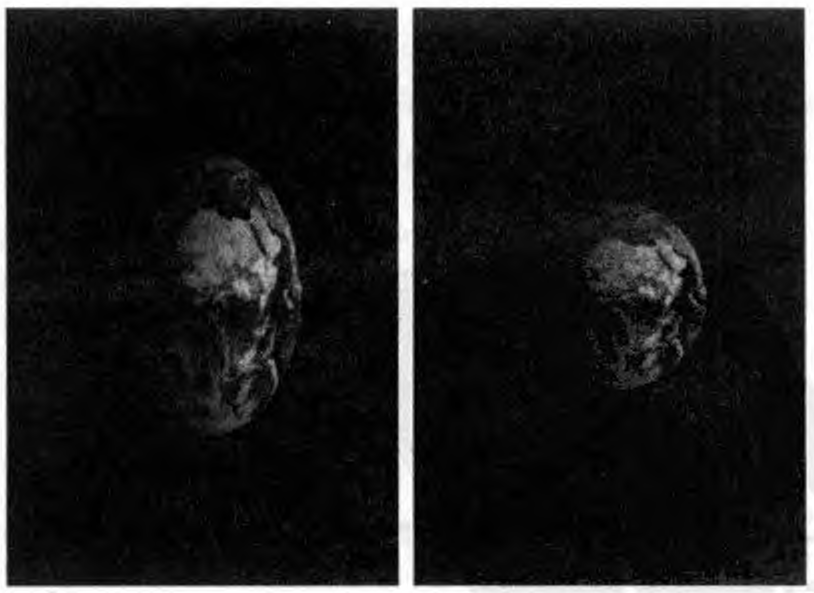


图 5-1　使用地球纹理的一个照亮的球体的两个例子

左图显示的地球看起来不像是一个球体，它看上去像是被压扁或者拉伸而成为一个鸡蛋的形状。拉伸渲染效果已经在第 2 章和第 3 章中的例子中提到过，它的发生是因为默认的 OpenGL ES 坐标系会映射沿着 X 轴的从 –1.0 到 1.0 的值到屏幕的宽度，并映射沿着 Y 轴的从 –1.0 到 1.0 的值到屏幕的高度。然而，屏幕并不是正方形，它的高要比宽的值大。一个半径为 0.5 的球体被拉伸显示在屏幕上，这是因为沿着 Y 轴的 0.5 的距离要比沿着 X 轴的 0.5 的距离覆盖更多的屏幕像素。

例子 OpenGLES_Ch5_1 和例子 OpenGLES_Ch5_2 几乎是相同的，但是 OpenGLES_

Ch5_2 使用了一个变换来矫正拉伸的渲染效果。

在 OpenGLES_Ch5_1 中的 OpenGLES_Ch5_1ViewController.m 文件中包含了 sphere.h 文件，这个文件包含球体的顶点位置、法向量和纹理坐标。在 sphere.h 中的数据是由一个 Perl 语言脚本生成的，这个脚本可以解析 Wavefront .obj 文件并输出对应的 C 语言代码。文件格式“.obj”是由 Wavefront Technologies 公司在 20 世纪 80 年代末开发和推广的。这个文件格式仍然流行是因为它用软件解析起来很容易。本章例子使用的 sphere.obj 数据文件是使用免费开源的 3D 内容创建工具 Blender 创建的，这个工具可以在 http://www.blender.org/ 上获得。sphere.obj 数据文件和用于解析它的 Perl 脚本都在本章例子的 Resources 目录里，这个例子可以从 http://opengles.cosmicthump.com/learning-opengl-es-sample-code/Examples.zip 上获得。

注意 截止到现在的所有例子都是硬编码的顶点数据。除了类似 -1、0 和 1 的最简单常量之外，硬编码其他的任何东西通常并不是一个好主意。这些例子提供了与数据简单显示使用的代码相似的顶点数据定义代码，不过第 4 章中只用 8 个顶点描述一个三角锥的例子已经达到了硬编码的实际极限。在例子 OpenGLES_Ch5_1 和 OpenGLES_Ch5_2 中的球体有 1944 个顶点。想象一下硬编码那么多的数据，即使你不需要全部键入。一些 3D 场景由数百万个顶点组成。实际上，类似 Blender 或者免费的谷歌 SketchUp 应用（http://sketchup.google.com/）的 3D 编辑器会在嵌入式应用之后读取的文件中生成大量的顶点数据。sphere.h 文件是一个折中方法，它包含了工具生成的数据，但被格式化为能够被直接编译的 C 代码。

下面的 OpenGLES_Ch5_1ViewController.m 文件结合了迄今为止介绍过的很多 OpenGL ES 概念。“-viewDidLoad”方法添加了使用一个深度缓存和多个顶点缓存的代码，参见粗体显示的代码：

```
////////////////////////////////////////////////////////////////////////
// Called when the view controller's view is loaded
// Perform initialization before the view is asked to draw
- (void)viewDidLoad
{
   [super viewDidLoad];

   // Verify the type of view created automatically by the
   // Interface Builder storyboard
   GLKView *view = (GLKView *)self.view;
   NSAssert([view isKindOfClass:[GLKView class]],
      @"View controller's view is not a GLKView");

   view.drawableDepthFormat = GLKViewDrawableDepthFormat16;
```

```
    // Create an OpenGL ES 2.0 context and provide it to the
    // view
    view.context = [[AGLKContext alloc]
       initWithAPI:kEAGLRenderingAPIOpenGLES2];

    // Make the new context current
    [AGLKContext setCurrentContext:view.context];

    // Create a base effect that provides standard OpenGL ES 2.0
    // shading language programs and set constants to be used for
    // all subsequent rendering
    self.baseEffect = [[GLKBaseEffect alloc] init];

    // Configure a light to simulate the Sun
self.baseEffect.light0.enabled = GL_TRUE;
self.baseEffect.light0.diffuseColor = GLKVector4Make(
    0.7f, // Red
    0.7f, // Green
    0.7f, // Blue
    1.0f);// Alpha
self.baseEffect.light0.ambientColor = GLKVector4Make(
    0.2f, // Red
    0.2f, // Green
    0.2f, // Blue
    1.0f);// Alpha
self.baseEffect.light0.position = GLKVector4Make(
    1.0f, // Off to the right of the Earth
    0.0f,
   -0.8f, // A bit in front of the Earth
    0.0f);

// Setup texture
CGImageRef imageRef =
    [[UIImage imageNamed:@"Earth512x256.jpg"] CGImage];

GLKTextureInfo *textureInfo = [GLKTextureLoader
    textureWithCGImage:imageRef
    options:[NSDictionary dictionaryWithObjectsAndKeys:
       [NSNumber numberWithBool:YES],
       GLKTextureLoaderOriginBottomLeft, nil]
    error:NULL];

self.baseEffect.texture2d0.name = textureInfo.name;
self.baseEffect.texture2d0.target = textureInfo.target;

// Set the background color stored in the current context
((AGLKContext *)view.context).clearColor = GLKVector4Make(
    0.0f, // Red
```

```
    0.0f, // Green
    0.0f, // Blue
    1.0f);// Alpha

// Create vertex buffers containing vertices to draw
self.vertexPositionBuffer = [[AGLKVertexAttribArrayBuffer alloc]
    initWithAttribStride:(3 * sizeof(GLfloat))
    numberOfVertices:sizeof(sphereVerts) / (3 * sizeof(GLfloat))
    data:sphereVerts
    usage:GL_STATIC_DRAW];
self.vertexNormalBuffer = [[AGLKVertexAttribArrayBuffer alloc]
    initWithAttribStride:(3 * sizeof(GLfloat))
    numberOfVertices:sizeof(sphereNormals) / (3 * sizeof(GLfloat))
       data:sphereNormals
       usage:GL_STATIC_DRAW];
    self.vertexTextureCoordBuffer = [[AGLKVertexAttribArrayBuffer alloc]
       initWithAttribStride:(2 * sizeof(GLfloat))
       numberOfVertices:sizeof(sphereTexCoords) / (2 * sizeof(GLfloat))
       data:sphereTexCoords
       usage:GL_STATIC_DRAW];

    [((AGLKContext *)view.context) enable:GL_DEPTH_TEST];
}
```

与之前把所有的顶点元素交错保存在一个缓存中的例子不同，在例子 OpenGLES_Ch5_1 中使用了多个顶点属性数组缓存。当所有的顶点属性靠在一起驻留在内存中时，大部分 GPU 可以最佳化执行。GPU 可能会在一个内存操作中读取所需的所有值。然而，用来生成在 sphere.h 文件中声明的数据的脚本会保存顶点位置、法向量和纹理坐标到不同的数组中，因此，如在前面的“-viewDidLoad”实现中以粗体标注的代码显示的一样，保存数据到不同的缓存是最容易的。当一些顶点元素频繁变换而剩下的保持不变时，使用不同的缓存有时可能会产生一个性能优势。例如，如果纹理坐标从来不改变但是顶点位置频繁改变，那么把顶点位置保存到一个使用 GL_DYNAMIC_DRAW 提示的缓存中，把纹理坐标保存在另一个使用 GL_STATIC_DRAW 提示的缓存中，这样应该是最佳化的。第 2 章在介绍 glBufferData() 函数时，讲解了 GL_DYNAMIC_DRAW 和 GL_STATIC_DRAW。

“-viewDidLoad”的实现最后开启了片元深度测试。如果你好奇想看看在不使用深度缓存的情况下在渲染的过程中会发生什么，请注释掉“[((AGLKContext *)view.context) enable:GL_DEPTH_TEST];”一行并重建。

使用多个顶点属性数组缓存提供的属性所做的绘制与以前的例子会有一点儿不同。为 AGLKContext 的“-clear:”方法指定 GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT 来在同一时间清除深度缓存和像素颜色渲染缓存。AGLKContext 的“-clear:”方法会调用 glClear() 函数。在“-viewDidLoad”中创建的每个 AGLKVertexAttribArrayBuffer

实例是为绘制准备的。准备好调用 glBindBuffer()、glEnableVertexAttribArray() 和 glVertexAttribPointer() 函数。最后，一个被添加到本例的 AGLKVertexAttribArrayBuffer 类的类方法“+drawPreparedArraysWithMode:startVertexIndex:numberOfVertices:”会调用 glDrawArrays() 函数，这个函数会绘制顶点，使用从每个开启属性的缓存指针收集来的数据。

```
////////////////////////////////////////////////////////////////
// GLKView delegate method: Called by the view controller's view
// whenever Cocoa Touch asks the view controller's view to
// draw itself. (In this case, render into a frame buffer that
// shares memory with a Core Animation Layer)
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect
{
   [self.baseEffect prepareToDraw];

   // Clear back frame buffer (erase previous drawing)
   [(AGLKContext *)view.context
      clear:GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT];

   [self.vertexPositionBuffer
      prepareToDrawWithAttrib:GLKVertexAttribPosition
      numberOfCoordinates:3
      attribOffset:0
      shouldEnable:YES];
   [self.vertexNormalBuffer
      prepareToDrawWithAttrib:GLKVertexAttribNormal
      numberOfCoordinates:3
      attribOffset:0
      shouldEnable:YES];
   [self.vertexTextureCoordBuffer
      prepareToDrawWithAttrib:GLKVertexAttribTexCoord0
      numberOfCoordinates:2
      attribOffset:0
      shouldEnable:YES];

   // Draw triangles using vertices in the prepared vertex
   // buffers
   [AGLKVertexAttribArrayBuffer
      drawPreparedArraysWithMode:GL_TRIANGLES
      startVertexIndex:0
      numberOfVertices:sphereNumVerts];
}
```

通过把下面的两个粗体显示的代码语句添加到“-glkView:drawInRect:”方法的末尾附近，OpenGLES_Ch5_2 修改了 OpenGLES_Ch5_1。

```
// Scale the Y coordinate based on the aspect ratio of the
// view's Layer which matches the screen aspect ratio for
// this example
const GLfloat  aspectRatio =
   (GLfloat)view.drawableWidth / (GLfloat)view.drawableHeight;

self.baseEffect.transform.projectionMatrix =
   GLKMatrix4MakeScale(1.0f, aspectRatio, 1.0f);

// Draw triangles using vertices in the prepared vertex
// buffers
[AGLKVertexAttribArrayBuffer
   drawPreparedArraysWithMode:GL_TRIANGLES
      startVertexIndex:0
      numberOfVertices:sphereNumVerts];
}
```

GLKit 的 GLKMatrix4MakeScale() 函数创建了一个基础的变换矩阵（transformation matrix）。这个函数接收三个参数，这三个参数会改变坐标系的三个轴的相对单位长度。指定一个值 1.0 意味着没有变化。OpenGLES_Ch5_2 会保持沿着 X 轴和 Z 轴的单位长度不改变，但是设置沿着 Y 轴的单位长度为这个表达式 (GLfloat)view.drawableWidth / (GLfloat)view.drawableHeight 的值，这个表达式会计算屏幕纵横比。drawableWidth 属性以像素为单位提供了视图的像素颜色渲染缓存的宽度，这个宽度与视图的全屏 Core Animation 层相匹配。drawableHeight 属性以像素为单位提供了高度。屏幕纵横比为宽度除以高度。沿着屏幕的 X 轴的可用像素的数量与沿着屏幕的 Y 轴的可用像素的数量是不一样的。通过纵横比缩放 Y 轴以抵消在 OpenGLES_Ch5_1 中看到的拉伸效果。

注意 缩放对灯光有一个潜在的影响。缩放会作用到所有的几何图形，包括法向量。在一个法向量被缩放以后，它可能就不再是一个单位向量，因此就不能与 OpenGL ES 灯光模拟一起正确工作。由 GLKit 的 GLKBaseEffect 类生成的 Shading Language 程序会按需正规化法向量。对于 OpenGL ES 1.x 来说，调用 glEnable(GL_NORMALIZE) 函数会告诉 GPU 让所有的法向量变回单位长度，在它们被缩放以后。

在 5.4 节中，当为渲染一个场景而改变视点时，会讲解变换矩阵的作用。顶点属性、矢量和矩阵这三个概念一起构成了理解 3D 图形的关键。

5.3 深入探讨不用 GLKit 添加深度缓存

在 iOS 5 引入 GLKit 之前，开发者需要创建与 GLKView 相似的一个 UIView 的子

类。回顾一下 2.4 节，例子 OpenGLES_Ch2_2 创建了 AGLKView 类以复制 GLKit 的 GLKView 类的多个功能；例子 OpenGLES_Ch5_3 通过复制它的深度缓存功能，类似地探索了 GLKView 是怎么工作的。几乎对于所有用途来说，基于苹果优化的、测试的、面向未来的 GLKView 的实现都是最好的，而不是任何应用特定的类。如果你没有兴趣继续深入学习 GLKView，可以随意跳过这一节。

下面是不用 GLKit 的辅助而管理一个 OpenGL ES 深度缓存所需的步骤：

1）Generate（生成）——告诉 OpenGL ES 为深度缓存生成一个独一无二的标识符。

2）Bind（绑定）——告诉 OpenGL ES 在接下来的操作中使用哪一个缓存。

3）Configure Storage（配置存储）——指定深度缓存的大小。

4）Attach（附加）——附加深度缓存到一个帧缓存。

OpenGLES_Ch5_3 产生的输出与 OpenGLES_Ch5_2 产生的输出相同，但是 OpenGLES_Ch5_3 使用了一个扩展自例子 OpenGLES_Ch2_2 的 AGLKView 版本，而不是 GLKit 的 GLKView。drawableDepthFormat 属性被添加到 AGLKView，同时还有一个用来存储 OpenGL ES 深度缓存标识符的实例变量。下面添加到 AGLKView 的“-layoutSubviews”方法的代码按需创建并配置了一个深度缓存来匹配视图的像素颜色渲染缓存的尺寸。

```
if (0 != depthRenderBuffer)
{
   glDeleteRenderbuffers(1, &depthRenderBuffer);
   depthRenderBuffer = 0;
}

GLint currentDrawableWidth = self.drawableWidth;
GLint currentDrawableHeight = self.drawableHeight;

if(self.drawableDepthFormat !=
   AGLKViewDrawableDepthFormatNone &&
   0 < currentDrawableWidth &&
   0 < currentDrawableHeight)
{
   glGenRenderbuffers(1, &depthRenderBuffer);  // Step 1
   glBindRenderbuffer(GL_RENDERBUFFER,         // Step 2
      depthRenderBuffer);
   glRenderbufferStorage(GL_RENDERBUFFER,      // Step 3
      GL_DEPTH_COMPONENT16,
      currentDrawableWidth,
      currentDrawableHeight);
   glFramebufferRenderbuffer(GL_FRAMEBUFFER,   // Step 4
      GL_DEPTH_ATTACHMENT,
      GL_RENDERBUFFER,
      depthRenderBuffer);
}
```

与一个 Core Animation 层分享内存的像素颜色渲染缓存在层调整大小时会自动调整大小。其他的缓存，例如深度缓存，并不会自动调整大小。每当一个 AGLKView 实例被重新调整大小时，与它相关联的层也会调整大小并且视图的“-layoutSubviews”方法被调用。AGLKView 实现了“-layoutSubviews”来删除任何现存的深度缓存，并创建一个新的与像素颜色渲染缓存的新尺寸相匹配的深度缓存。

5.4 变换

从数学上说，变换就是在两个坐标系之间转换顶点坐标。每个坐标系都是相对于其他的参照坐标系定义的。对于 OpenGL ES 来说，最终的参照坐标系是在一个像素颜色渲染缓存中的像素位置的 2D 数组。默认 3D 坐标系是相对于最终参考坐标系定义的，能够用来将 –1.0 到 1.0 范围内的 X 坐标值转换为像素颜色渲染缓存对应宽度处的像素位置，将 –1.0 到 1.0 范围内的 Y 坐标值转换为像素颜色渲染缓存对应高度处的像素位置。除了 Z 坐标值在 –1.0 到 1.0 范围外的顶点不会绘制以外，Z 坐标值不会影响默认 3D 坐标系跟像素坐标之间的转换。

注意 在渲染的过程中，GPU 会转换点、线和三角形的顶点数据为着色的片元。当前（实际上是在 GPU 处理顶点数据时）坐标系决定了片元在像素颜色渲染缓存和深度缓存中的位置。当前坐标系因此决定了每个片元会显示在屏幕的哪里，并使从任意视点渲染场景成为可能。

程序能够相对于默认 3D 坐标系（作为一个参考）定义一个新的坐标系。可以相对于其他坐标系定义任意数量的坐标系。在任何情况下，在一个新坐标系中的顶点位置可以被转换回参考的坐标系。因此，可以转换在任意坐标系中的任意顶点位置为像素颜色渲染缓存中的一个位置。

变换看起来需要大量的数学计算才能在坐标系之间进行转换，但是线性代数再次提供了解决方案。任意数量任意顺序的基本变换都能够被捕获并保存在一个简单的 4 乘 4 的浮点值矩阵中。一个矩阵定义一个坐标系。第 11 章会介绍 GPU 在坐标系之间进行顶点转换时要用到的实际方程式。

矩阵计算几乎完全使用加法和乘法，并且在现代 GPU 上执行得非常快。这是个好事，因为每次一个场景被渲染时，GPU 都会变换所有的顶点，并且在一个场景中常常含有几十万个顶点。

5.4.1 基本变换

只存在四个基本变换：平移（translation）、旋转（rotation）、缩放（scale）和透视

(perspective)。基本变换联合决定了在一个新坐标系中的每一个顶点位置是怎么转换为参考坐标系中的一个位置的。四个基本变换足以产生变化无穷的坐标系。

每个基本变换对应于矩阵的一个简单变化。定义一个与参考坐标系相同的坐标系的矩阵叫做单位矩阵（identity matrix）。任意两个矩阵可以在一个级联（concatenation）操作中结合起来以产生一个新矩阵，这个新矩阵包含了两个矩阵的所有变换。实际上，每个基本变换会产生一个简单的矩阵，然后把这个简单的矩阵与当前变换矩阵连接起来以产生一个新的当前矩阵。

注意　“矩阵级联”有时又被称为“矩阵乘法”（matrix multiplication）。术语“矩阵乘法”还适用于使用一个矩阵把一个矢量或者顶点从一个坐标系变换到另一个的操作。“级联”是一个更精确的术语，它描绘了两个矩阵的结合，这个结合产生了包含这两个源矩阵的所有信息的第三个矩阵。

1. 平移

通过相对于参考坐标系的原点移动新坐标系的原点，平移定义了一个新的坐标系。平移不会影响坐标轴的单位长度，平移不会改变坐标轴相对于参考坐标系的方向。图 5-2 描述了一个新坐标系，这个坐标系是对参考坐标系的平移。

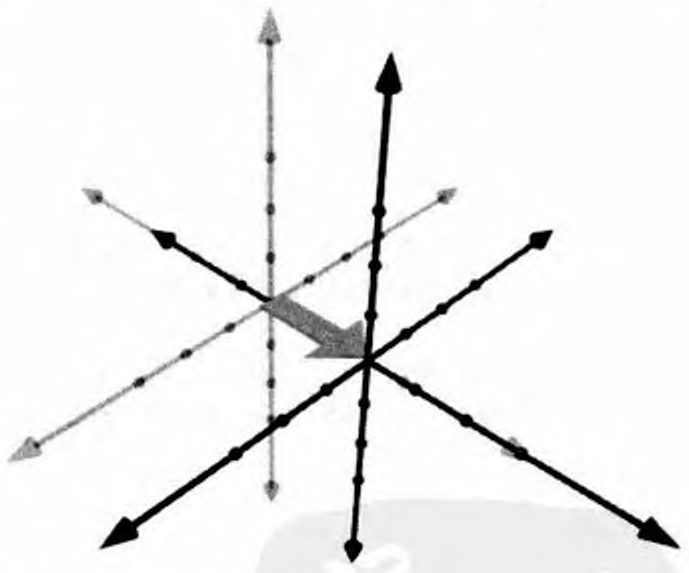


图 5-2　平移

GLKit 提供了 GLKMatrix4MakeTranslation（float x，float y，float z）函数，这个函数通过平移一个单位矩阵来返回一个定义了坐标系的新矩阵。x、y 和 z 参数指定了新坐标系的原点沿着当前参考坐标系的每个轴移动的单位数。函数 GLKMatrix4Translate(GLKMatrix4 matrix, float x, float y, float z) 通过平移作为参数传入的矩阵来返回定义了一个坐标系的新矩阵。从概念上看，GLKMatrix4Translate() 会返回参数矩阵与 GLKMatrix4MakeTranslation() 产生的新矩阵的级联。

```
GLKMatrix4 GLKMatrix4Translate(GLKMatrix4 matrix,
   float x, float y, float z)
{
   return GLKMatrix4Multiply(matrix,
      GLKMatrix4MakeTranslation(x, y, z));
}
```

2. 旋转

旋转是通过相对于参考坐标系坐标轴的方向旋转新坐标系的坐标轴来定义一个新坐标系。旋转的坐标系会与参考坐标系使用同一个原点。旋转不会影响坐标轴的单位长度，只有坐标轴的方向会发生变化。图 5-3 描绘了一个新坐标系，这个坐标系是由旋转一个参考坐标系来定义的。

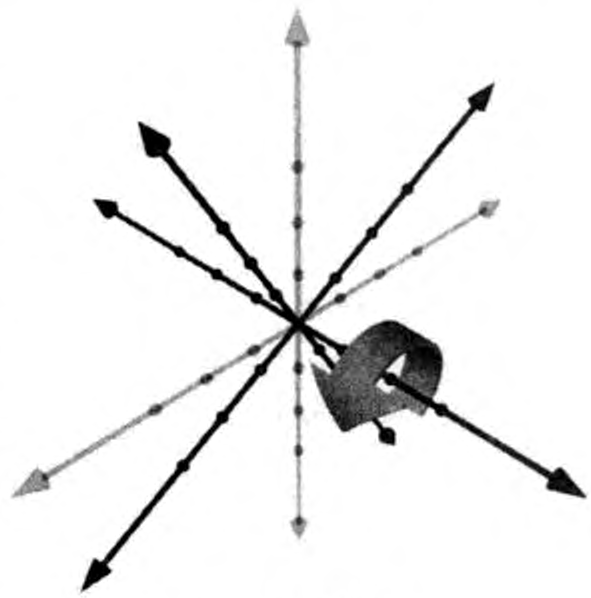


图 5-3　旋转

GLKit 提供了 GLKMatrix4MakeRotation(float angleRadians, float x, float y, float z) 函数，这个函数通过旋转一个单位矩阵来返回定义了一个坐标系的新矩阵。angleRadians 参数指定了要旋转的弧度数。使用 GLKMathDegreesToRadians() 函数可以把角度转换成弧度。x、y 和 z 参数用于指定当前坐标系的哪一个轴作为旋转的轮毂。例如，代码 GLKMatrix4MakeRotation(GLKMathDegreesToRadians(30.0), 1.0, 0.0, 0.0); 会沿着一个特定的坐标系的 X 轴旋转 30 度来产生一个新的坐标系。GLKMatrix4Rotate(GLKMatrix4 matrix, float angleRadians, float x, float y, float z) 函数会通过旋转一个作为参数传入的矩阵来返回定义了一个坐标系的新矩阵。从概念上看，GLKMatrix4Rotate() 会返回参数矩阵与 GLKMatrix4MakeRotation() 产生的新矩阵的级联。

```
GLKMatrix4 GLKMatrix4Rotate(GLKMatrix4 matrix,
   float angleRadians, float x, float y, float z)
{
   return GLKMatrix4Multiply(matrix,
      GLKMatrix4MakeRotation(angleRadians, x, y, z));
}
```

3. 缩放

缩放是通过相对于参考坐标系的坐标轴的单位长度改变新坐标系的坐标轴的单位长度来定义一个新坐标系。缩放的坐标系与参考坐标系使用同一个原点，坐标轴的方向通常不会改变。不过，通过一个负值所做的缩放会翻转坐标轴的方向。例如，如果增加一个坐标轴的值通常代表方向向上，那么使用一个负值缩放那个坐标轴后，再增加这个坐标轴的值就代表方向向下。图 5-4 描述了一个通过缩放参考坐标系来定义的新坐标系。

图 5-4　缩放

GLKit 提供了 GLKMatrix4MakeScale(float x, float y, float z) 函数，这个函数会通过扩大或者缩小一个单位矩阵的任意坐标轴的单位长度来返回一个定义了坐标系的矩阵。x、y 和 z 参数指定了用来扩大或者缩小每个轴的单位长度的因数。GLKMatrix4-Scale(GLKMatrix4 matrix, float x, float y, float z) 函数通过按指定的因数缩放作为参数传入的矩阵来返回一个定义了坐标系的新矩阵。从概念上看，GLKMatrix4Scale() 会返回参数矩阵与 GLKMatrix4MakeScale() 产生的新矩阵的级联。

```
GLKMatrix4 GLKMatrix4Scale(GLKMatrix4 matrix,
   float x, float y, float z)
{
   return GLKMatrix4Multiply(matrix,
      GLKMatrix4MakeScale(x, y, z));
}
```

例如，函数 GLKMatrix4Scale(matrix, 2.0, 1.0, 2.0) 会返回一个新矩阵，这个矩阵拉伸了由参数矩阵定义的坐标系的 X 和 Z 轴的单位长度。

4. 透视

透视是通过相对于参考坐标系的坐标轴的单位长度多样化新坐标系的坐标轴的单位长度来定义一个新的坐标系。透视不会改变坐标轴的方向或者原点，但是坐标轴的每个单位离原点越远长度越短。这个效果会让在远处的物体比离原点近的物体显得更小。图 5-5 描述了通过相对于参考坐标系的透视定义的一个新坐标系。

GLKit 提供了 GLKMatrix4MakeFrustum(float left, float right, float bottom, float top, float nearVal, float farVal) 函数，这个函数会透视一个单位矩阵来返回一个定义了坐标系的新矩阵。透视坐标系的形状是一个类似角锥体的平截头体。平截头体会在 5.6 节做详细解释。

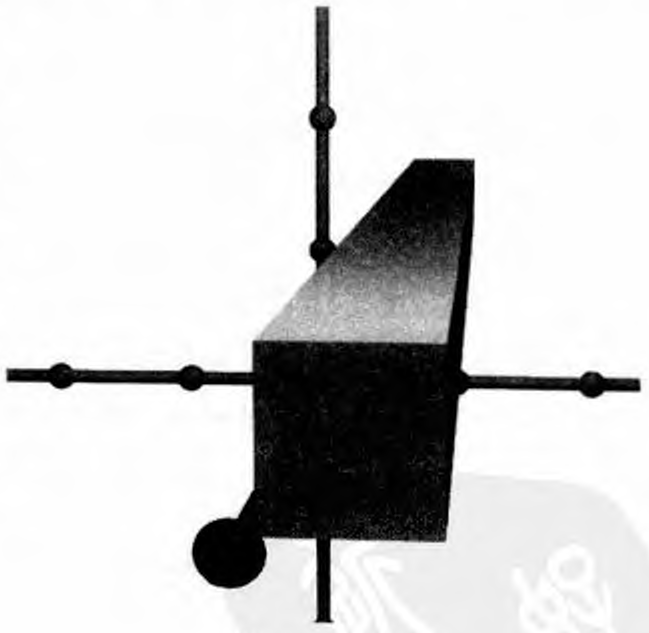


图 5-5　透视

5.4.2　顺序很重要

结合四大基本变换可以定义新坐标系，不过重要的是变换的顺序。平移在旋转的前面还是后面会产生不同的坐标系。缩放在旋转的前面还是后面会产生不同的坐标系。先旋转再平移最后缩放与先平移再旋转最后缩放是不同的，诸如此类。例子 OpenGLES_Ch5_4 为测试变换提供了用户界面控件，参见图 5-6。

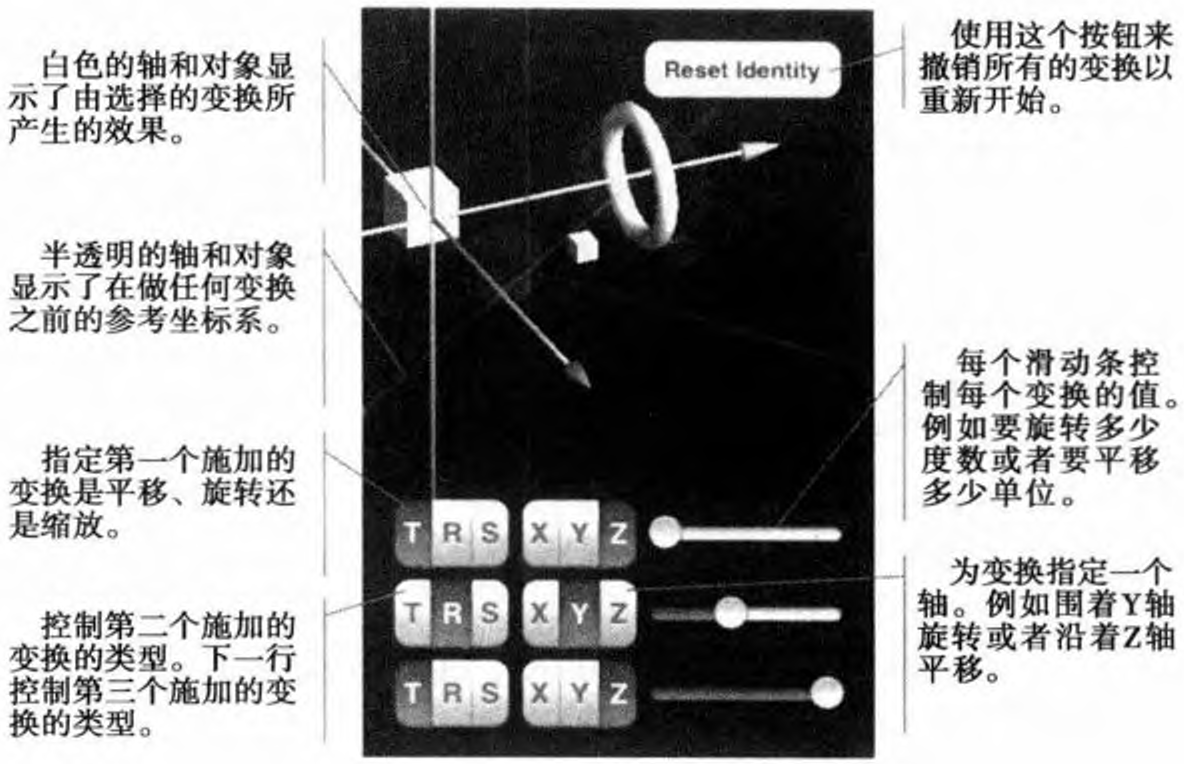


图 5-6　例子 OpenGLES_Ch5_4

5.4.3　projectionMatrix 和 modelviewMatrix

GLKBaseEffect 的 transform 属性是一个 GLKEffectPropertyTransform 类型的实例并为支持常见的操作保存了三个不同的矩阵。其中的两个矩阵，projectionMatrix 和 modelviewMatrix 分别定义了一个用于整个场景的坐标系和一个用于控制对象（又叫做场景内模型）显示位置的坐标系。GLKBaseEffect 会级联 modelviewMatrix 和 projectionMatrix 以产生一个 modelviewProjectionMatrix 矩阵，这个矩阵会把对象顶点完全地变换到 OpenGL ES 默认坐标系中。默认坐标系直接映射到像素颜色渲染缓存中的片元位置。图 5-7 显示了理论上的一系列变换，以及在渲染过程中为变换顶点所产生的坐标系。

图 5-7　顶点变换矩阵和结果坐标系

OpenGLES_Ch5_4 应用会设置 baseEffect 的 projectionMatrix 来定义一个正射投影，正射投影会在 5.6 节中进行讲解。

```
const GLfloat  aspectRatio =
      (GLfloat)view.drawableWidth / (GLfloat)view.drawableHeight;

   self.baseEffect.transform.projectionMatrix =
      GLKMatrix4MakeOrtho(
         -0.5 * aspectRatio,
         0.5 * aspectRatio,
         -0.5,
         0.5,
         -5.0,
         5.0);
```

在 OpenGLES_Ch5_4ViewController 的“-viewDidLoad”方法中初始化的 modelviewMatrix 矩阵会使用一个明显的右上视角来渲染对象，参见图 5-6。

```
GLKMatrix4 modelviewMatrix = GLKMatrix4MakeRotation(
   GLKMathDegreesToRadians(30.0f),
   1.0,  // Rotate about X axis
   0.0,
   0.0);
modelviewMatrix = GLKMatrix4Rotate(
   modelviewMatrix,
   GLKMathDegreesToRadians(-30.0f),
   0.0,
   1.0,  // Rotate about Y axis
   0.0);
modelviewMatrix = GLKMatrix4Translate(
   modelviewMatrix,
   -0.25,// Translate apparent position left and into the screen
   0.0,
   -0.20);
```

在用户移动图 5-6 中显示的滑动条的同时，会更新每个用户用来控制变换的值。在例子 OpenGLES_Ch5_4 重绘时，下面的代码会重新计算 modelviewMatrix。首先，保存 baseEffect 的 modelviewMatrix 的当前值，以便可以在后面加载它。通过按特定的顺序连接当前矩阵与用户控制的三个变换矩阵，这个例子创建了一个新的矩阵。修改顺序就会修改变换的最终结果。接着，把新矩阵设置给 baseEffect 的 modelviewMatrix。

```
// Save the current Modelview matrix
   GLKMatrix4 sevedModelviewMatrix =
      self.baseEffect.transform.modelviewMatrix;

   // Combine all of the user chosen transforms in order
   GLKMatrix4 newModelviewMatrix =
      GLKMatrix4Multiply(sevedModelviewMatrix,
      SceneMatrixForTransform(
         transform1Type,
         transform1Axis,
         transform1Value));
```

```
newModelviewMatrix =
   GLKMatrix4Multiply(newModelviewMatrix,
   SceneMatrixForTransform(
      transform2Type,
      transform2Axis,
      transform2Value));
newModelviewMatrix =
   GLKMatrix4Multiply(newModelviewMatrix,
   SceneMatrixForTransform(
      transform3Type,
      transform3Axis,
      transform3Value));

// Set the Modelview matrix for drawing
self.baseEffect.transform.modelviewMatrix = newModelviewMatrix;
```

在用户控制的变换结合到 modelviewMatrix 后，为绘制准备好 baseEffect，最后用一个白色的漫反射光绘制场景中的对象。

```
// Make the light white
self.baseEffect.light0.diffuseColor = GLKVector4Make(
   1.0f, // Red
   1.0f, // Green
   1.0f, // Blue
   1.0f);// Alpha

[self.baseEffect prepareToDraw];

// Draw triangles using vertices in the prepared vertex
// buffers
[AGLKVertexAttribArrayBuffer
   drawPreparedArraysWithMode:GL_TRIANGLES
   startVertexIndex:0
   numberOfVertices:lowPolyAxesAndModels2NumVerts];
```

重新设置 modelviewMatrix 为保存的值，灯光颜色改成黄色，再次绘制相同的对象。场景中的黄色对象提供了一个参考，以便可以清楚地观察用户控制的变换会怎么影响在最终的像素颜色渲染缓存中的白色对象的哪些片元。

```
// Restore the saved Modelview matrix
self.baseEffect.transform.modelviewMatrix =
   sevedModelviewMatrix;

// Change the light color
self.baseEffect.light0.diffuseColor = GLKVector4Make(
   1.0f, // Red
   1.0f, // Green
   0.0f, // Blue
   0.3f);// Alpha
```

```
[self.baseEffect prepareToDraw];

// Draw triangles using vertices in the prepared vertex
// buffers
[AGLKVertexAttribArrayBuffer
   drawPreparedArraysWithMode:GL_TRIANGLES
   startVertexIndex:0
   numberOfVertices:lowPolyAxesAndModels2NumVerts];
```

例子 OpenGLES_Ch5_4 通过复制矩阵到一个临时的变量来保存并恢复 modelview-Matrix。GLKit 提供了一个方便的数据类型 GLKMatrixStack，还提供了一个用来向栈数据结构保存矩阵的函数集合。堆栈是一个后进先出的数据结构，它可以方便地存储某个程序可能需要恢复的矩阵。GLKMatrixStack 会实现一个 4×4 矩阵的堆栈。GLKMatrixStackPush() 函数会复制最顶部的矩阵到堆栈的顶部。GLKit 为修改矩阵堆栈顶部的矩阵提供了一个综合的函数集合，包括 GLKMatrixStackMultiplyMatrix4() 函数，这个函数是其他函数的基础。GLKMatrixStackGetMatrix4() 函数会返回最顶部的矩阵。GLKMatrixStackPop() 函数会移除堆栈最顶部的项，并把前一个顶部矩阵恢复到最顶部位置。

应用会推入一个新矩阵到堆栈顶部，操作并使用它在 OpenGL ES 中渲染几何图形，然后把它弹出堆栈并把上一个矩阵恢复到堆栈的顶部。从例子 OpenGLES_Ch5_5 开始，本书的很多例子都使用了 GLKMatrixStack。

5.4.4　textureMatrix

桌面 OpenGL 和 OpenGL ES 1.x 包含三个内建的矩阵堆栈。前两个保存投影矩阵和 model-view 矩阵。第三个是纹理矩阵，S 和 T 坐标系的纹理与顶点的 U 和 V 坐标之间有一个映射，纹理矩阵会向这个映射施加变换。纹理映射是在第 3 章讲解的。在写作本书时，GLKBaseEffect 的 transform 属性还没有使用或者提供一个 textureMatrix。因此，例子 OpenGLES_Ch5_5 用一个新的 AGLKTextureTransfromBaseEffect 类扩展了 GLKit 的 GLKBaseEffect 类，用来显示使用纹理矩阵可能会产生的一些效果。正如本书中的其他 AGLK 类，如果某天 GLKit 添加了一个 textureMatrix 属性，那么应该优先使用苹果的实现。

注意　在例子 OpenGLES_Ch5_5 中的 AGLKTextureTransformBaseEffect 类使用了自定义的 OpenGL ES 2.0 的 Shading Language 程序来实现与 OpenGL ES 1.x 的内置支持相似的纹理矩阵操作。看看 AGLKTextureTransformBaseEffect 的实现，观察一下 Shading Language 程序实际是怎么使用的，找出实现 GLKBaseEffec 的线索。GLKit 的存在是为了让你可以使用常用的 OpenGL ES 功能实现 3D 应用，而不需要知道 Shading Language 的所有细节。

下面来自 OpenGLES_Ch5_5 的摘录显示了 OpenGLES_Ch5_5ViewController 的“-glkView:drawInRect:”方法，以此来演示利用 AGLKTextureTransformBaseEffect 的 textureMatrix2d1 属性和一个 GLKMatrixStack 所做的纹理坐标变换。当使用多重纹理时，textureMatrix2d1 属性会保存用来变换第二个纹理的纹理坐标的矩阵。多重纹理效果是在 3.5 节介绍的。AGLKTexture TransformBaseEffect 还提供了一个 textureMatrix2d0 属性，用于使用多重纹理时第一个纹理的纹理坐标的变换。

```
GLKMatrixStackPush(self.textureMatrixStack);

   // Scale and rotate about the center of the texture
   GLKMatrixStackTranslate(
      self.textureMatrixStack,
      0.5, 0.5, 0.0);
   GLKMatrixStackScale(
      self.textureMatrixStack,
      textureScaleFactor, textureScaleFactor, 1.0);
   GLKMatrixStackRotate(   // Rotate about Z axis
      self.textureMatrixStack,
      GLKMathDegreesToRadians(textureAngle),
      0.0, 0.0, 1.0);
   GLKMatrixStackTranslate(
      self.textureMatrixStack,
      -0.5, -0.5, 0.0);

   self.baseEffect.textureMatrix2d1 =
      GLKMatrixStackGetMatrix4(self.textureMatrixStack);

   [self.baseEffect prepareToDrawMultitextures];

   // Draw triangles using currently bound vertex buffer
   [self.vertexBuffer drawArrayWithMode:GL_TRIANGLES
      startVertexIndex:0
      numberOfVertices:sizeof(vertices) / sizeof(SceneVertex)];

GLKMatrixStackPop(self.textureMatrixStack);

self.baseEffect.textureMatrix2d1 =
      GLKMatrixStackGetMatrix4(self.textureMatrixStack);
```

上述代码中的 textureScaleFactor 和 textureAngle 变量是由用户界面中的滑动条所控制的。运行例子并通过设置滑动条来测试以观察最终的纹理效果。把对于 self.baseEffect.textureMatrix2d1 的引用换成 self.baseEffect.textureMatrix2d0，以此来改变变换的纹理映射。

5.5　复合变换手册

本节会讲解一些常见的复合变换序列。

5.5.1　倾斜

倾斜（skew）是一个复合变换，这个变换产生了不再相互垂直的坐标轴。立方体会变成带有梯形面的盒子。

1）围着一个轴旋转。

2）施加不均匀的缩放，比如缩放 X 轴但不缩放 Y 轴或者 Z 轴。

3）沿着在第一步中使用的轴做反向旋转。

想要在例子 OpenGLES_Ch5_4 中看到倾斜效果，首先围着 Y 轴旋转 45°，然后沿着 X 轴尽量放大，最后围着 Y 轴回转。测试并试着对齐白色变换轴和半透明的参考轴。

5.5.2　围着一个点旋转

旋转和缩放常常围绕当前坐标系的原点发生。想象一下，太阳就是我们太阳系的假想坐标系的原点。地球围绕着太阳旋转，但也沿一条自己的轴旋转。围绕一个点而不是原点旋转的解决方案很简单。

1）平移到所需的旋转中心。

2）施加所需的旋转。

3）使用与第 1 步相反的平移值平移回来。

5.4.4 节中关于 textureMatrix 模式的例子需要围着纹理的中心而不是它的左下角旋转纹理坐标系。试着注释掉例子 OpenGLES_Ch5_5 中的包含平移的代码行。重建并重新运行例子以观察结果。

为了围绕例子 OpenGLES_Ch5_4 中的圆环形状的中心旋转，设置第一个滑动条沿着 Z 轴平移并把滑动条一直移动到最左边。使用第二个滑动条来围着 Y 轴旋转。设置第 3 个滑动条沿着 Z 轴平移并把滑动条一直移动到最右边。保持第一个和最后一个滑动条不改变，利用中间的滑动条测试不同的旋转量。

5.5.3　围着一个点缩放

围绕任意一个点缩放与围绕一个点旋转相似。

1）平移到所需的缩放中心。

2）施加想要的缩放。

3）使用与第 1 步相反的平移值平移回来。

5.6 透视和平截头体

OpenGL ES 使用一个叫做视域（viewing volume）的几何图形来决定一个场景生成的哪些片元将会显示在最终的渲染结果中。处于视域范围之外的几何图形会被剔除（clipped），这意味着它会被丢弃。视域有时也被称为投影（projection）。在本章之前，在例子中使用的视域一直是一个立方体，这个立方体包括了在 X 轴上从 –1.0 到 1.0，Y 轴上从 –1.0 到 1.0，Z 轴上 –1.0 到 1.0 范围内的所有顶点。这就是当 projectionMatrix 是一个单位矩阵时的默认投影结果。

一个立方体或者矩形视域叫做一个正射投影（orthographic projection）。利用一个正射投影，视点与每个位置之间的距离对于投影毫无影响。GLKit 的 GLKMatrix4-MakeOrtho(GLfloat left, GLfloat right, GLfloat bottom, GLfloat top, GLfloat near, GLfloat far) 函数会返回一个矩阵，这个矩阵定义了一个由 left 和 right，bottom 和 top，near 和 far 所界定的矩形视域。图 5-8 的左图描绘了由函数 GLKMatrix4MakeOrtho(-1.0, 1.0, -1.0, 1.0, 0.0, -6.0) 返回的矩阵所定义的视域。GLKMatrix4MakeOrtho 函数返回的矩阵通常与当前 projectionMatrix 级联来定义视域。

当使用透视投影（perspective projection）时，视域不再是矩形。它变成了角锥体被切去顶端之后的形状——称为“平截头体”，如图 5-8 的右图所描绘的。当投影到 2D 像素颜色渲染缓存中后，离视点越远的对象越小，但是接近视点的对象仍然很大。GLKit 的 GLKMatrix4MakeFrustum() 函数与 GLKMatrix4MakeOrtho() 函数的参数相同，并且会创建一个包含透视的新变换矩阵。这个新矩阵通常会与当前 projectionMatrix 级联来定义视域。

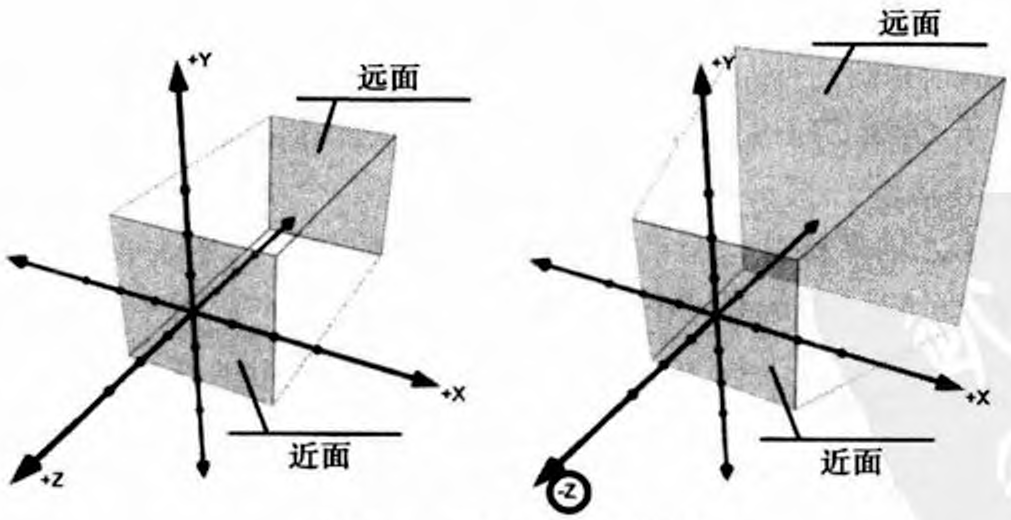


图 5-8　正射和透视视域

图 5-8 中的两个视域中的半透明着色区域代表了每个视域的近面。在使用 GLKMatrix4MakeFrustum() 函数时，对于近面有一些限制，但在使用 GLKMatrix4MakeOrtho() 函数时却没有。平截头体不能用 Z 位置值小于 0 的近面来定义。即使是为近面使用很小

的正的 Z 值也会有问题。从近面距离到远面距离之间的位置范围会映射为深度缓存中的深度值。当 GPU 计算要被保存在深度缓存中的值时，使用大幅度或者过度小的近面距离都会产生数学舍入误差。

注意　对 GLKMatrix4MakeFrustum() 函数的限制产生了一个小问题：OpenGL ES 默认为一个指入屏幕的负的 Z 坐标轴，但是 GLKMatrix4MakeFrustum() 总会产生一个指入屏幕的带有正的 Z 坐标轴的视域。GLKMatrix4MakeFrustum() 函数翻转了 Z 坐标轴的符号。

使用视域的另一个关键是要认识到，正射和透视投影都是由站在位置 {0，0，0} 并向下俯视 Z 轴的观察者的视点产生的。本章开头声称变换使任意视点成为可能，但是实际上，视点是恒定的。变换会施加到场景中的所有顶点上，并会投影这些顶点到视域的近面上。从概念上看，当假设的观察者的视点保持不变时，OpenGL ES 会变换所有的几何图形。整个世界移动的错觉只是看起来像是由视点变化产生的。

例子 OpenGLES_Ch5_6 演示了如何使用 GLKMatrix4MakeFrustum() 函数来产生透视变换。这个例子模拟了地球沿着自己的轴旋转，同时月球绕地球轨道而行。当月球接近视点时，它看起来很大，但是当它逐渐远离视点时变得越来越小。例子 OpenGLES_Ch5_6 还演示了除了倾斜之外的所有常见复合变换。

5.7　小结

本章介绍了深度缓存、变换、GLKit 的矩阵、GLKit 的矩阵堆栈和视域。实际上本章中没有例子会改变保存在顶点缓存中的顶点的成分。顶点位置保持不变。作为替代，变换改变了顶点被映射到 2D 像素颜色渲染缓存的方式。例子 OpenGLES_Ch5_6 通过动态地改变当前 modelviewMatrix 产生了一些简单的动画。第 6 章会进一步讲解基于变换的各种动画可能，并会介绍多个实现动画和特效的其他方法。

第6章 动 画

随着时间改变一个渲染场景的可视内容就会产生图形动画。变化坐标系会产生一个场景内的移动效果。改变顶点数据会产生类似水波荡漾和随风飘摇的旗帜的效果。变化颜色和灯光可以模拟出很多电影氛围。改变纹理坐标系可以动画化几何体的纹理映射过程，变化纹理数据可以产生类似电影的效果。

动画的关键是随着时间的流逝管理时间并执行应用逻辑。GLKit 的 GLKViewController 类提供了灵活的时间支持。控制器的“-update”方法会自动按照配置周期率被调用。紧随“-update”函数的调用，控制器的视图会被重绘。最重要的是，GLKViewController 会让“-update”函数的调用和视图的绘制与显示刷新同步。

显示刷新率通常依赖于嵌入式设备的硬件，显示刷新率确定了屏幕像素的颜色每秒可以变化的最大次数。因此，GLKViewController 的“-update”、GLKView 的“-drawInRect:”和 GLKView 的 delegate（委托）方法“-glkView:drawInRect:”为应用实现动画提供了完美的触发器。让一个渲染场景的可视内容的变化速度快于显示刷新率没有任何意义，用户只能看到在显示刷新之前的最后一次更新。

GLKViewController 的 timeSinceLastUpdate、timeSinceLastDraw、timeSinceLastResume 和 timeSinceFirstResume 属性会以秒和秒的一部分为度量保存时间间隔。时间信息控制动画。例如，如果一个几何对象每秒移动模拟的 5 厘米，那么 timeSinceLastUpdate 时间乘以 5 厘米就确定了对象在这次位置更新中需要移动多远。GLKViewController 的 paused 属性控制更新是否暂停，framesDisplayed 属性保存了自应用启动以来显示更新的总次数。

本章的所有例子都使用 GLKViewController 来实现时间以及与 iOS 功能的整合，比如自动显示方向变化。如果你好奇 GLKViewController 是怎么工作的，请浏览下在例子 OpenGLES_Ch2_2 和 OpenGLES_Ch5_3 中包含的自定义 AGLKViewController 类，这个类使用苹果的一个 Core Animation 类（即 CADisplayLink 类）部分地重新实现了 GLKViewController 的时间功能。如第 2 章讲解的，Core Animation 最终通过在硬件显示刷新时切换 OpenGL ES 的前帧缓存和后帧缓存来控制渲染场景的显示。Core Animation 的 CADisplayLink 会计算 GLKViewController 所使用的时间信息。

6.1 场景内移动：例子 OpenGLES_Ch6_1

动画通常包含两种情况，对象相对于用户的视点移动或者用户的视点相对于对象的位置变化。第 5 章中的例子 OpenGLES_Ch5_6 演示了改变渲染场景中的视点的一个方法。这个例子会在正射投影和透视投影之间切换。与很多 3D 图像概念相似，视点存在于一个抽象的数学领域中。实际上，当前的 OpenGL ES 视点永远不会改变。矩阵定义了几何对象（如三角形）是怎么映射到像素颜色渲染缓存中的片元位置的。改变矩阵进而改变映射，最终创建了视点变化的错觉。其实，观察者并没有移动，只是计算出来的每个可视对象的位置相对于观察者发生了变化。

本章的第一个例子 OpenGLES_Ch6_1 模拟了一个游乐园碰碰车。碰碰车在墙壁之间来回弹或者相互碰撞。图 6-1 用两个屏幕截图展示了一个场景的不同视点。第一个屏幕截图显示的是观察者在碰碰车上方向下看的效果。第二个屏幕截图显示了从一个活动中的碰碰车中的视点所观察到的场景。

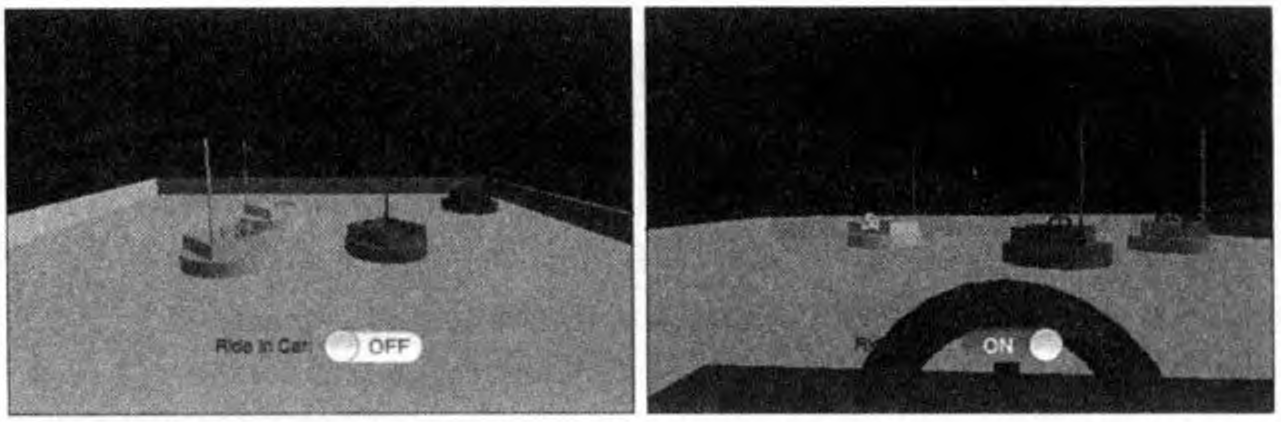


图 6-1 显示了不同视点的屏幕截图

应用运行时，在示例 OpenGLES_Ch6_1 中，位置或者法向量等顶点属性都不会发生变化。碰碰车在溜冰场内来回弹的动画完全是由矩阵所代表的坐标系随着时间变化产生的。事实上，每辆车都处在它自己的本地坐标系的 {0，0，0} 位置。当碰碰车的本地坐标系相对于用来渲染其他碰碰车和溜冰场的坐标系发生变化时，碰碰车就好像在场景内来回移动。

例子 OpenGLES_Ch6_1 是基于之前章节中介绍的多个概念建立的。当碰碰车彼此相对移动或者相对于观察者移动的时候就产生了动画。当用户切换到“乘坐在车上”（Ride in Car）模式时就产生了视点自身的动画。

6.1.1 看向一个特定的 3D 位置

OpenGLES_Ch6_1 使用 GLKit 的 GLKMatrix4MakeLookAt() 函数来计算一个几乎可以代表任意所需视点的变换矩阵。GLKMatrix4MakeLookAt() 函数与实用函数 gluLookAt() 相似，gluLookAt() 函数来自桌面版 OpenGL，但并不是 OpenGL ES 的一部分。GLKMatrix4MakeLookAt() 等便利函数重新创建了 OpenGL ES 丢弃的很多功能，

这些函数是在 iOS 应用中使用 GLKit 的最好的原因之一。

GLKMatrix4MakeLookAt() 有 6 个参数，参见下面的函数声明。前三个参数指定观察者眼睛的 {x，y，z} 位置，接下来的 3 个参数指定观察者正在看向的 {x，y，z} 位置。GLKMatrix4MakeLookAt() 函数会计算并返回一个 model-view 矩阵，这个矩阵会对齐从眼睛的位置到看向的位置之间的矢量与当前视域的中心线。如果眼睛的位置与看向的位置相同，函数 GLKMatrix4MakeLookAt() 就不会产生有效的结果。第 5 章介绍过视域和 model-view 矩阵。

```
GLKMatrix4 GLKMatrix4MakeLookAt(
   float eyeX, float eyeY, float eyeZ,
   float lookatX, float lookatY, float lookatZ,
   float upX, float upY, float upZ);
```

GLKMatrix4MakeLookAt() 函数的最后三个参数指定了“上”方向矢量的 {x，y，z} 元素。改变“上”方向与倾斜观察者头部的效果相同。

注意 “上”方向可以是任意矢量，但是 GLKMatrix4MakeLookAt() 的实现所使用的数学计算不能产生一个有效的直接顺着“上”矢量看的视点。这个限制的存在是因为当直接向“上”或者向“下”看时，GLKMatrix4MakeLookAt() 函数使用的数学计算会试图计算 90° 角的正切，而这在数学上是未定义的。这个“未定义”的现象还发生在现实世界中，例如，当机械陀螺仪碰到“万象结锁定”并且产生摇摆不可靠的数据时。但是，存在一个巧妙的数学解决方案——四元法，第 11 章将做简短介绍，网址 http://en.wikipedia.org/wiki/Quaternion 也有相关介绍。GLKit 包含了一个 GLKQuaternion 数据类型和一些用来操作它的函数。维基百科讲述了 William Rowan Hamilton 爵士在 1843 年发现四元法的故事，这是在数学史上最有名和最有趣的故事之一。

OpenGLES_Ch6_1 的 OpenGLES_Ch6_1ViewController 使用命名为 eyePosition 和 lookAtPosition 的属性保存当前的眼睛位置和看向的位置。控制器的“-glkView:drawInRect:”方法使用如下代码设置当前的 model-view 矩阵。

```
// Set the modelview matrix to match current eye and look-at
// positions
self.baseEffect.transform.modelviewMatrix =
   GLKMatrix4MakeLookAt(
      self.eyePosition.x,
      self.eyePosition.y,
      self.eyePosition.z,
      self.lookAtPosition.x,
      self.lookAtPosition.y,
      self.lookAtPosition.z,
      0, 1, 0);
```

在3D应用中，活动中的视点叫做“第一人称”视点，因为它呈现了一个观察者站在应用环境内部时的场景。另一种视点称为“第三人称”，它模拟了从一个活动之外的有利位置从上往下看的视图。添加一个新的方法“-updatePointOfView”到例子的OpenGLES_Ch6_1ViewController来设置“目标”眼睛位置和看向的位置，如下代码所示。目标位置在动画中起了一个重要的作用，下一节会详细讲解。

```
- (void)updatePointOfView
{
   if(!self.shouldUseFirstPersonPOV)
   {  // Set the target point of view to arbitrary "third person"
      // perspective
      self.targetEyePosition = GLKVector3Make(6.00, 1.75, 0.0);
      self.targetLookAtPosition = GLKVector3Make(0, 0.5, 0);
   }
   else
   {  // Set the target point of view to a position within the
      // last car and facing the direction of the car's motion.
      SceneCar *viewerCar = [cars lastObject];

      // Set the new target position up a bit from center of
      // car
      self.targetEyePosition = GLKVector3Make(
         viewerCar.position.x,
         viewerCar.position.y + 0.45f,  // up an arbitrary amount
         viewerCar.position.z);

      // Look from eye position in direction of motion
      self.targetLookAtPosition = GLKVector3Add(
         eyePosition,
         viewerCar.velocity);
   }
}
```

第三人称视点会把观察者的眼睛置于溜冰场的侧上部并看向溜冰场中央稍微向上的位置。第三人称眼睛和看向的位置是任意角度的并且不会发生变化。相比之下，第一人称视点会随着观察者所乘坐的碰碰车而移动和转向。眼睛位置被设置为碰碰车当前位置的正上方，看向位置是在碰碰车前面的碰碰车行驶方向上的一个点。在前面的代码中，viewerCar.velocity是一个用于指定碰碰车行驶方向的带有方向的矢量，并且会指定碰碰车移动的速度。把viewerCar.velocity与eyePosition相加会计算出碰碰车前方的一个位置。

6.1.2 使用时间

类似从第三人称视点瞬间转向第一人称视点的不和谐视觉变化会让用户失去方向感。OpenGLES_Ch6_1ViewController的“-update”方法为用户变化渲染视点提供了一

个平滑的过渡动画。这个动画是使用一个低通滤波器来逐渐减少当前视点与用户选择的视点之间的差异产生的。低通滤波器会反复逐渐地改变计算出来的值，并且必须调用很多次才能产生一个明显的效果。之所以叫做“低通”是因为对于正在被过滤的值来说，低频的、长期的变化会有一个明显的影响，而高频变化的影响甚微。下面来自 OpenGLES_Ch6_1 的 SceneCar.m 文件的代码实现了低通滤波器。

```
//////////////////////////////////////////////////////////////////////////
// This function returns a value between target and current. Call
// this function repeatedly to asymptotically return values closer
// to target: "ease in" to the target value.
GLfloat SceneScalarSlowLowPassFilter(
   NSTimeInterval elapsed,    // seconds elapsed since last call
   GLfloat target,            // target value to approach
   GLfloat current)           // current value
{  // Constant 4.0 is an arbitrarily "small" factor
   return current + (4.0 * elapsed * (target - current));
}


//////////////////////////////////////////////////////////////////////////
// This function returns a vector between target and current.
// Call repeatedly to asymptotically return vectors closer
// to target: "ease in" to the target value.
GLKVector3 SceneVector3SlowLowPassFilter(
   NSTimeInterval elapsed,    // seconds elapsed since last call
   GLKVector3 target,         // target value to approach
   GLKVector3 current)        // current value
{
   return GLKVector3Make(
      SceneScalarSlowLowPassFilter(elapsed, target.x, current.x),
      SceneScalarSlowLowPassFilter(elapsed, target.y, current.y),
      SceneScalarSlowLowPassFilter(elapsed, target.z, current.z));
}
```

低通滤波器是通过反复调用来工作的，每次调用会返回一个更接近“目标”值的新的当前值。每当调用 OpenGLES_Ch6_1ViewController 的“-update”方法时，也会调用低通滤波器函数并且当前视点变得更加接近目标视点，直至当前视点最终与目标视点一致。

与低通滤波器类似的函数会让动画更流畅。OpenGLES_Ch6_1 也会在改变碰碰车的方向时使用过滤器。当碰碰车从墙壁弹回并转向时，碰碰车并不会瞬间转向新的方向，而是先让目标方向变为新的方向，然后碰碰车的当前方向逐步更新直到与目标方向一致。几乎所有的 3D 模拟都受益于这样或者那样的过滤器。

OpenGLES_Ch6_1 向之前章节中开发的类所在的组添加了五个新类：SceneMesh、SceneModel、SceneCar、SceneCarModel 和 SceneRinkModel。SceneMesh 类的存在是为

了管理大量的顶点数据以及 GPU 控制的内存数据的坐标转换。网格（mesh）就是共享顶点或者边，同时用于定义 3D 图形的三角形的一个集合。SceneModel 类会绘制全部或者部分的网格。一个单独的模型可能由多个网格组成，多个模型可能共用相同的网格。模型代表了汽车、山脉或人物等 3D 对象，这些 3D 对象的形状是由网格定义的。模型聚合了描绘 3D 对象所需的所有网格。SceneCar 类封装了每个碰碰车的当前位置、速度、颜色、偏航角和模型。偏航（yaw）是来自轮船和航空的一个术语，代表了围绕垂直轴的旋转度，在这里是围绕 Y 轴。偏航定义了碰碰车的方向并且会随着时间变化而让碰碰车面向它移动的方向。SceneModel 的子类 SceneCarModel 封装了一个碰碰车形状的网格。每个 SceneCar 实例都使用同一个 SceneCarModel 实例。如果想要碰碰车拥有不同的外形，比如有的看起来像卡车，有的看起来像飞机，那么就需要每个 SceneCar 实例使用一个不同的模型。最后 SceneModel 的子类 SceneRinkModel 封装了代表溜冰场的墙壁和地面的网格。

SceneCar 实现了“-drawWithBaseEffect:”方法，这个方法会设置当前材质的颜色以匹配碰碰车的颜色，平移 model-view 坐标系到碰碰车的当前位置，旋转坐标系以匹配碰碰车的当前偏航角，并绘制碰碰车模型。平移和旋转已经在第 5 章中讲过。

```
//////////////////////////////////////////////////////////////////
// Draw the receiver: This method sets anEffect's current
// material color to the receiver's color, translates to the
// receiver's position, rotates to match the receiver's yaw
// angle, draws the receiver's model. This method restores the
// values of anEffect's properties to values in place when the
// method was called.
- (void)drawWithBaseEffect:(GLKBaseEffect *)anEffect;
{
   // Save effect attributes that will be changed
   GLKMatrix4  savedModelviewMatrix =
      anEffect.transform.modelviewMatrix;
   GLKVector4  savedDiffuseColor =
      anEffect.material.diffuseColor;
   GLKVector4  savedAmbientColor =
      anEffect.material.ambientColor;

   // Translate to the model's position
   anEffect.transform.modelviewMatrix =
      GLKMatrix4Translate(savedModelviewMatrix,
          position.x, position.y, position.z);

   // Rotate to match model's yaw angle (rotation about Y)
   anEffect.transform.modelviewMatrix =
      GLKMatrix4Rotate(anEffect.transform.modelviewMatrix,
          self.yawRadians,
          0.0, 1.0, 0.0);
```

```
    // Set the model's material color
    anEffect.material.diffuseColor = self.color;
    anEffect.material.ambientColor = self.color;

    [anEffect prepareToDraw];

    // Draw the model
    [model draw];

    // Restore saved attributes
    anEffect.transform.modelviewMatrix = savedModelviewMatrix;
    anEffect.material.diffuseColor = savedDiffuseColor;
    anEffect.material.ambientColor = savedAmbientColor;
}
```

本章中的所有例子都会重用 SceneMesh 类，并且除了一个之外其余所有的例子都会使用 SceneModel 类。在每个例子中的视图控制器都会分配并初始化 SceneMesh 和 SceneModel 实例。控制器的“-update”方法会被自动地按照显示刷新率调用，并且这个方法会更新变量以间接地控制绘图坐标系。然后，控制器的“-glkView:drawInRect:”方法会发送绘制模型和网格的消息。

6.2 动画化顶点数据

例子 OpenGLES_Ch6_2 会随着时间修改网格顶点属性来模拟一个随风飘扬的旗帜。图 6-2 是显示结果。网格动画产生的动态效果有时是必需的，但是在 3D 图形中网格动画并不常见，因为它需要在 GPU 控制的内存和 CPU 控制的内存之间频繁地复制顶点数据。坐标系的变化可以产生很多与顶点属性变化相同的效果，但是却不用复制所有的数据。例如，动画 3D 人物常常是由定义人物外形的复杂非矩形网格构建而成。当人物走、跑、跳时，网格内的顶点会保持不变，但是变换矩阵会让其看起来就像是人物的四肢通过关节和肌肉在移动，并且皮肤会相应拉伸。第 7 章会详细讲解模型。

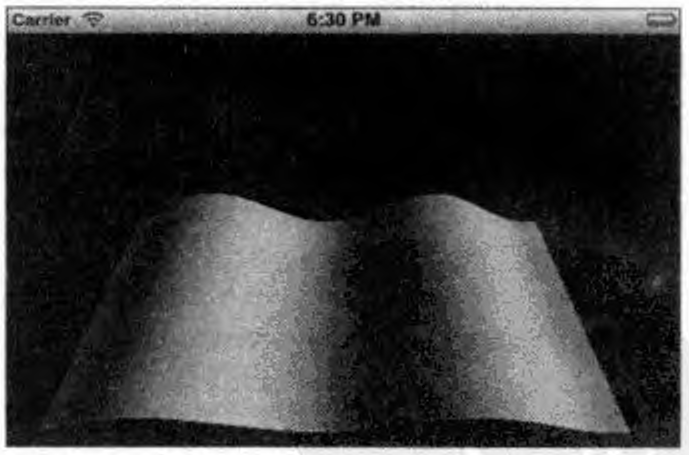


图 6-2 例子 OpenGLES_Ch6_2 实现的飘扬的旗帜的屏幕截图

当使用网格动画时，通过优化来减少需要复制的数据量通常意义非凡。为 GPU 提供三角形几何图形数据的最简单方法是指定所有三角形的三个顶点。但是，当绘制共边三角形时可以使用一个叫做三角形带（triangle strip）的优化方法。一个三角形带结合

了两个或者更多个相互连接的三角形。在这个带中的第一个三角形是由前三个顶点定义的，接下来的每个三角形会与带中上一个三角形共用两个顶点。图 6-3 显示了怎么用标识为 A 和 B 的两个三角形绘制一个正方形，但却只用标识为 0、1、2 和 3 的四个顶点。在图 6-3 中的两个三角形共用编号为 1、2 的顶点。

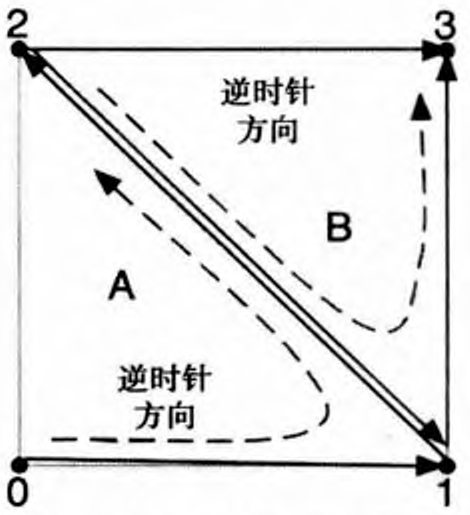


图 6-3 绘制一个由两个三角形组成的四边形需要四个顶点

共享顶点减少了保存几何图形信息所需要的内存容量，减少了 GPU 必须处理的顶点的总数量。为一个场景中的所有三角形分别指定 3 个顶点需要（3×*N*）个顶点（*N* 是三角形的数量）。而使用三角形带绘制相同的场景只需要（2×*N*）个顶点。

三角形和三角形带中的顶点顺序是不同的。OpenGL ES 使用顶点顺序来决定每个三角形的哪一面是前面。在一个 3D 场景中，每个三角形都有一个前面和一个后面。大部分情况下，你能够看到哪一面是没有关系的，但是 OpenGL ES 可以通过使用背面剔除（back face culling）模式来设置跳过一面或者另一面的渲染，背面剔除会在第 7 章介绍。按照惯例，首先根据第 4 章提到的右手原则，法向量指向前面。对于图 6-3 中的三角形 A 来说，法向量是使用顶点（0，1，2）并按照逆时针顺序计算出来的。如果顶点是按相反的顺序（0，2，1）处理的，那么结果法向量会指向相反的方向。

如果图 6-3 中的正方形是由两个不同的三角形绘制的，那么顶点的正确顺序是三角形（0，1，2）、三角形（2，1，3）。这样，两个三角形都是按照逆时针顺序处理的。但是，如果在图 6-3 中的正方形是用一个带有 4 个顶点的三角形带绘制的，即三角形带（0，1，2，3），那么 OpenGL ES 会以一个非常有意义的顺序解释三角形带的顶点：第一个三角形是按（0，1，2）的顺序渲染的，但是第二个三角形并不是按照（1，2，3）的顺序，而是按照（2，1，3）的顺序渲染的。图 6-4 显示了一个用来绘制五个相连的三角形的更加复杂的三角形带。在图 6-4 中的顶点是按照三角带（0，1，2，3，4，5，6）的顺序提交给 OpenGL ES 的，但是这些顶点是按照（0，1，2），（2，1，3），（2，3，4），（4，3，5），（4，5，6）的顺序处理的，这会赋予所有的五个三角形一个逆时针的顶点顺序。

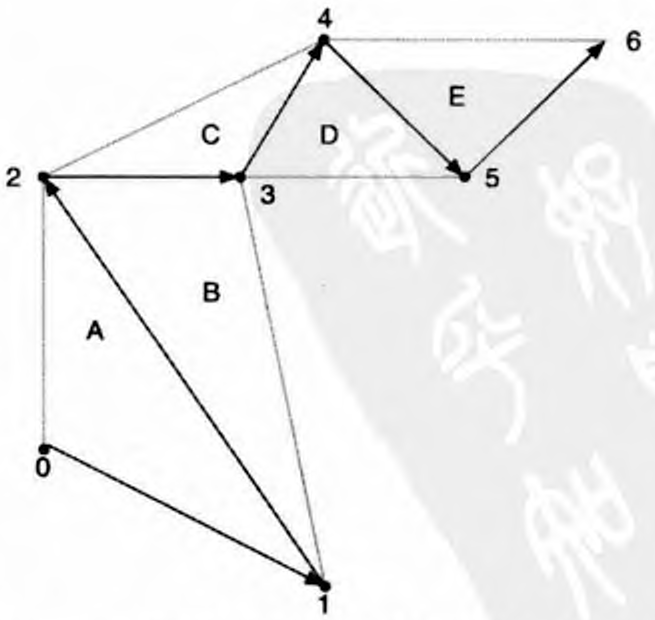


图 6-4 一个三角形带形成了一个复杂的图形

图 6-4 中的编号指出了顶点在一个顶点

缓存中的存储顺序。绘制图 6-4 中显示的三角形带是非常简单的，只需要绑定顶点数组缓存到当前 OpenGL ES 上下文并调用 glDrawArrays(GL_TRIANGLE_STRIP，0，7) 函数即可。

6.2.1 使用索引顶点

绘制类似在图 6-5 中显示的网格有一个技巧。当一个网格有多个列时，单独调用 glDrawArrays(GL_TRIANGLE_STRIP，…) 而不复制顶点是不可能的。有必要用单独三角形带来绘制每一列。但是在这个网格中的每列与上一列共用一半的顶点数据。例如，在图 6-5 中的标识为 5 的顶点是在第一个列中标识为 A、B 和 C 的三角形的一部分，也是在第二列中标识为 D、E 和 F 的三角形的一部分。用一个单独的三角形带绘制每个列需要复制在内存中的两个列之间的共享顶点。换句话说，同样的顶点数据既要保存在用来绘制第一个列的顶点缓存中，又要保存在绘制第二列的顶点缓存中。

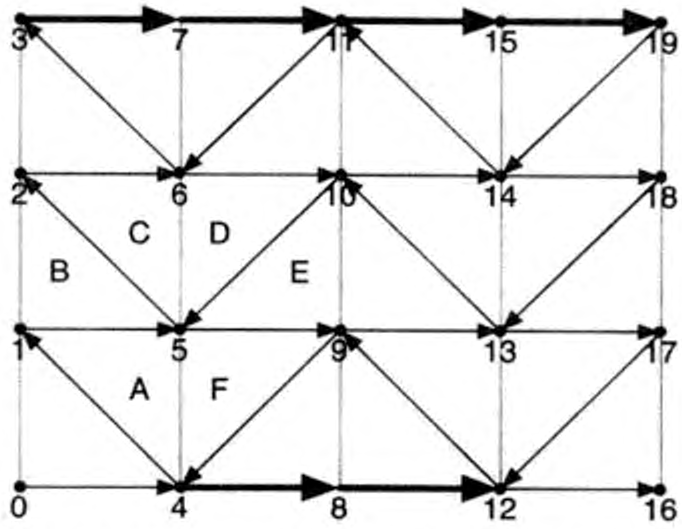


图 6-5 网格顶点和索引的几何关系

索引顶点提供了一个优化，这个优化可以消除奢侈的顶点数据复制。当使用索引顶点时，每个顶点只需要在内存中保存一次，无论有多少个三角形使用了这个顶点。

为了应用索引顶点，需要创建一个叫做元素数组缓存（element array buffer）的新缓存来保存顶点索引。在元素数组缓存中的每个值都通过顶点的位置确定了在一个单独的顶点属性数组缓存中的一个顶点。索引 0 标识在顶点属性数组缓存中的第一个顶点。索引 5 标识在顶点属性数组缓存中的第六个顶点（从 0 开始计数），以此类推。元素数组缓存的创建与其他缓存非常类似，参见下面的伪代码。

```
glGenBuffers(1, &indexBuffer);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBuffer);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, <size>,
      <address>, <usage_hint>);
```

按照用来绘制三角形带的参考顶点的顺序保存元素数组缓存内的顶点索引值。相同的索引值可以通过在元素数组缓存中重复这个值来使用任意次。

使用一个元素数组缓存绘制与使用 glDrawArrays() 函数绘制是相似的，不过使用的是 glDrawElements(GLenum mode, GLsizei count, GLenum type, const GLvoid *indices) 函数作为替代。glDrawElements() 函数的 mode 参数所支持的值包括：GL_POINTS、GL_LINE_STRIP、GL_LINE_LOOP、GL_LINES、GL_TRIANGLE_STRIP、GL_

TRIANGLE_FAN 和 GL_TRIANGLES。例子 OpenGLES_Ch6_2 使用的是 GL_RTIANGLE_STRIP 模式。网址 http://www.khronos.org/opengles/documentation/opengles1_0/html/glDrawElements.html 上有所有模式的帮助文档。glDrawElements() 的第二个参数指定了要使用的索引的数量。对于三角形带来说，索引的数量等于要绘制的三角形的数量加上 2。第三个参数指定了索引值的类型，必须是 GL_UNSIGNED_BYTE 或者 GL_UNSIGNED_SHORT。注意，当使用 GL_UNSIGNED_SHORT 类型时最多可以引用 65 536 个独一无二的索引，当使用 GL_UNSIGNED_BYTE 时只能引用 256 个独一无二的索引。glDrawElements() 的最后一个参数是一个指向包含这些索引的内存的指针。当使用一个元素数组缓存时，第四个参数实际上是一个元素数组缓存的字节偏移量，并且当起始于第一个索引时，其值应该等于 NULL。

使用 glDrawElements() 函数绘制网格有一个技巧：为了在一次 glDrawElements() 函数调用中绘制多个网格行，需要在每一列的末尾插入空三角形。图 6-5 显示了带有粗线的空三角形。下面的列表包含了绘制网格所需要的顶点索引，这些索引是按照它们在元素数组缓存中的保存顺序排列的。这个列表使用粗体突出了产生空三角形的索引。

```
{0, 4, 1, 5, 2, 6, 3, 7, 11, 6, 10, 5, 9, 4, 8, 12, 9, 13, 10, 14, 11, 15,
19, 14, 18, 13, 17, 12, 16}
```

顶点（3，7，11）、（4，8，12）和（11，15，19）所组成的三角形都没有面积并且在渲染时都不会产生片元。

这个使用空三角形的网格绘制方式可以在一次 glDrawElements() 函数调用中绘制最多带有 65 536 个不同顶点的网格。每列的那个空三角形的处理所浪费的 GPU 处理时间要远远低于为每列的绘制单独调用 glDrawElements() 函数所需要的额外处理时间。

6.2.2 OpenGLES_Ch6_2 示例

例子 OpenGLES_Ch6_2 创建了 OpenGLES_Ch6_1 中的 SceneMesh 类的子类 SceneAnimatedMesh。下面的代码摘录自 SceneMesh 并且与前面例子的视图控制器中的“-glkView:drawInRect:”方法的实现代码相似。如果有必要保存网格顶点属性，则会创建一个 AGLKVertexAttribArrayBuffer 实例。一个新的元素数组缓存被创建，并用索引初始化，同时被绑定到 OpenGL ES 的上下文，参见粗体代码。

```
//////////////////////////////////////////////////////////////////
// This method prepares the current OpenGL ES 2.0 context for
// drawing with the receiver's vertex attributes and indices.
- (void)prepareToDraw;
{
   if(nil == self.vertexAttributeBuffer &&
      0 < [self.vertexData length])
   {  // vertex attiributes haven't been sent to GPU yet
```

```
        self.vertexAttributeBuffer =
            [[AGLKVertexAttribArrayBuffer alloc]
            initWithAttribStride:sizeof(SceneMeshVertex)
            numberOfVertices:[self.vertexData length] /
               sizeof(SceneMeshVertex)
            data:[self.vertexData bytes]
            usage:GL_STATIC_DRAW];

        // No longer need local data storage
        self.vertexData = nil;
    }

    if(0 == indexBufferID && 0 < [self.indexData length])
    {  // Indices haven't been sent to GPU yet
        // Create an element array buffer for mesh indices
        glGenBuffers(1, &indexBufferID);
        NSAssert(0 != self.indexBufferID,
            @"Failed to generate element array buffer");

        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, self.indexBufferID);
        glBufferData(GL_ELEMENT_ARRAY_BUFFER,
            [self.indexData length],
            [self.indexData bytes],
            GL_STATIC_DRAW);

        // No longer need local index storage
        self.indexData = nil;
    }

    // Prepare vertex buffer for drawing
    [self.vertexAttributeBuffer
        prepareToDrawWithAttrib:GLKVertexAttribPosition
        numberOfCoordinates:3
        attribOffset:offsetof(SceneMeshVertex, position)
        shouldEnable:YES];

    [self.vertexAttributeBuffer
        prepareToDrawWithAttrib:GLKVertexAttribNormal
        numberOfCoordinates:3
        attribOffset:offsetof(SceneMeshVertex, normal)
        shouldEnable:YES];

    [self.vertexAttributeBuffer
        prepareToDrawWithAttrib:GLKVertexAttribTexCoord0
        numberOfCoordinates:2
        attribOffset:offsetof(SceneMeshVertex, texCoords0)
        shouldEnable:YES];

    // Bind the element array buffer (indices)
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferID);
}
```

SceneAnimatedMesh 类提供了下面的“-drawEntireMesh”方法，这个方法会调用 glDrawElements() 函数，而不是前面的例子中使用的 glDrawArrays() 函数。

```
////////////////////////////////////////////////////////////////
// Draw the entire mesh after it has been prepared for drawing
- (void)drawEntireMesh;
{
   // Draw triangles using vertices in the prepared vertex
   // buffers and indices from the bound element array buffer
   glDrawElements(GL_TRIANGLE_STRIP,
      NUM_MESH_INDICES,
      GL_UNSIGNED_SHORT,
      NULL);
}
```

这个例子通过随着时间改变网格顶点的 Y 坐标来产生网格动画。碧波荡漾的水波效果是通过在视图控制器的“-update”方法中以一个固定的频率调用下面的 SceneAnimateMesh 方法产生的。

```
////////////////////////////////////////////////////////////////
// This method modifies vertex positions and recalculates
// normals. The vertex attribute array is reinitialized with the
// modified vertex data in this class' -draw method.
- (void)updateMeshWithElapsedTime:(NSTimeInterval)anInterval;
{
   int    currentRow;
   int    currentColumn;

   // For each position along +X axis of mesh
   for(currentColumn = 0; currentColumn < NUM_MESH_COLUMNS;
      currentColumn++)
   {
      const GLfloat   phaseOffset = 2.0f * anInterval;
      const GLfloat   phase = 4.0 * currentColumn /
        (float)NUM_MESH_COLUMNS;
      const GLfloat   yOffset = 2.0 *
         sinf(M_PI * (phase + phaseOffset));

      // For each position along -Z axis of mesh
      for(currentRow = 0; currentRow < NUM_MESH_ROWS; currentRow++)
      {
         mesh[currentColumn][currentRow].position.y =
            yOffset;
      }
   }

   SceneMeshUpdateNormals(mesh);

   [self makeDynamicAndUpdateWithVertices:&mesh[0][0]
      numberOfVertices:sizeof(mesh) / sizeof(SceneMeshVertex)];
}
```

在前面方法中的粗体代码会设置每个顶点的 Y 坐标的值为一个正弦三角函数的值，这个正弦三角函数的值基于被修改的顶点的当前列和一个时间间隔。每次重新计算值时，时间间隔就会变化，这产生了一个移动的波纹效果。使用正弦只是因为它能产生一个形象有趣的重复样式。使用正弦之外的其他函数也是可以的。试试不同的方程式，比如 yOffset = 2.0 * cosf(phase * phase + phaseOffset) 或者 yOffset = 2.0 * sinf(M_PI * (sinf(phase) + phaseOffset))，来产生不同的波。

每次顶点位置发生变化时，顶点法向量也必须被重新计算。对于 SceneMeshUpdate Normals() 的调用会使用一个与第 4 章介绍的相似的算法实现这一点。计算出网格中的每个三角形的面法线，同时设置每个顶点的法向量为包含这个顶点的所有三角形的面法线向量的一个平均值。平均面法线向量会产生一个光滑的而不是多面的灯光效果。

最后，调用“makeDynamicAndUpdateWithVertices: numberOfVertices:”方法以向 GPU 控制的内存发送更新的顶点属性，并设置网格的使用提示为 GL_DYNAMIC_DRAW 来表明顶点属性可能会频繁更新。

注意 网格动画需要消耗内存带宽来从 CPU 控制的内存向 GPU 控制的内存复制更新的顶点数据。最大限度地减少复制发生的总量是非常重要的，因为内存带宽是嵌入式系统的头号瓶颈。避免复制的一个方法是直接在 GPU 控制的内存中使用运行在 GPU 上的一个 OpenGL ES 2.0 Shading Language 程序来计算新的顶点位置。7.4 节将详细讲解这个程序。

6.3 动画化颜色和灯光：例子 OpenGLES_Ch6_3

图 6-6 是例子 OpenGLES_Ch6_3 的屏幕截图。随着时间重定位 OpenGL ES 灯光或者让 OpenGL ES 灯光指向不同的位置会产生视觉效果。每个灯光的颜色和强度可以变化。例子 OpenGLES_Ch6_3 会添加灯光动画到例子 OpenGLES_Ch6_2 生成的飘扬旗帜上。

例子 OpenGLES_Ch6_3 会使用第 4 章简单提到的“聚光灯”。聚光灯是在一个圆锥体范围内散发光线而不是各个方向上散发光线的位置光源。除了有一个位置，每个聚光灯还有一个 spotDirection、一个 spotCutoff 角度和一个 spotExponent，如图 6-7 所示。spotDirection 是一个用来定义圆锥体内的光线照射方向的矢量。spotCutoff 角度是一个用度计量的角度，这个角度决定了灯光圆锥体的宽度。spotExponent 决定了灯光从聚光灯的中心到边缘变暗的速度。

GLKit 的 GLKBaseEffect 和 GLKEffectPropertyLight 类一起实现了聚光灯。GLKit 提供了两个不同的 OpenGL ES 灯光实现方式。设置 GLKBaseEffect 的 lightingType 属性为 GLKLightingTypePerVertex 会告诉 GLKit 去计算在几何图形中的每个顶点的光线

值，并在顶点之间插值。设置 lightingType 为 GLKLightingTypePerPixel 会产生一个更高质量的渲染结果，但这需要为每个片元重新计算光线效果，而这比每顶点计算要耗费更多的 GPU 计算量。不为聚光灯做片元计算，光照范围的边缘就会出现锯齿。在 OpenGLES_Ch6_3ViewController 的“-viewDidLoad”方法里将 lightingType 设置为 GLKLightingTypePerVertex 就会看到锯齿。

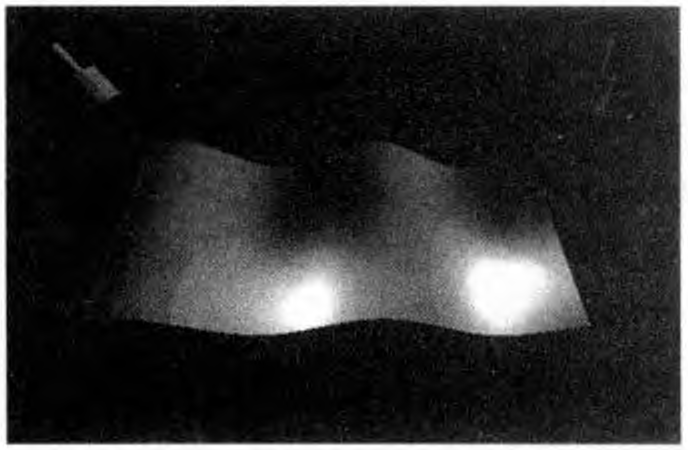


图 6-6 活动的聚光灯照亮了飘扬的旗帜

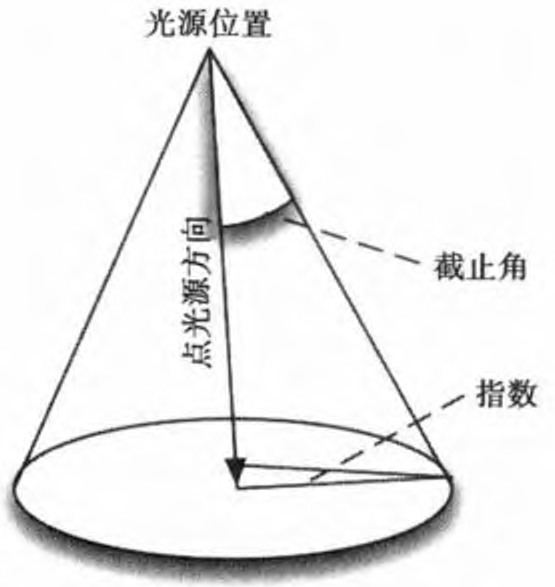


图 6-7 聚光灯属性

例子 OpenGLES_Ch6_3 和本章接下来的例子会使用 GLKit 的 GLKBaseEffect 类的自定义子类 AGLKTextureTransformBaseEffect。在例子 OpenGLES_Ch6_3 中这两个类会产生相同的渲染结果。AGLKTextureTransformBaseEffect 类的实现强调的是坐标系变换，这个坐标系变换是指本章后面将会讲解的灯光位置和方向的坐标系变换。另外，后面的例子会用到在写作本书时 GLKBaseEffect 还不支持的纹理效果。现在利用 AGLKTextureTransformBaseEffect 可以最大限度地减少例子 OpenGLES_Ch6_3 和后面的例子之间的不同。

注意 AGLKTextureTransformBaseEffect 类的实现重新创建了 GLKit 的 GLKBaseEffect 类的子集，这个子集包含了聚光灯和定向灯。AGLKTextureTransformBaseEffect 类使用的 Shading Language 程序是精心制作的，但是仍然不能提供 GLKBaseEffect 类的所有灯光功能。跟其他的“AGLK”类一样，最好尽可能使用苹果支持的版本。浏览一下 AGLKTextureTransformBaseEffect 类的实现，感受一下 GLKit 的实现方式（利用 OpenGL ES 2.0），以及 Shading Language 程序的潜在限制。

聚光灯是在 OpenGLES_Ch6_3ViewController 的“-viewDidLoad”方法中配置的。首先，这个方法会设置总体的灯光模式。

```
// Use high quality lighting (important for spot lights)
self.baseEffect.lightingType = GLKLightingTypePerPixel;
self.baseEffect.lightModelTwoSided = GL_FALSE;
self.baseEffect.lightModelAmbientColor = GLKVector4Make(
   0.6f, // Red
   0.6f, // Green
   0.6f, // Blue
   1.0f);// Alpha
```

双面灯光可以让几何体对象的前面和背面使用不同的材质。前面就是法向量的方向与之背离的面。光线与材质的相互作用决定了片元的颜色，因此使用不同的材质有可能产生令人印象深刻的效果。在写作本书时，GLKBaseEffect 在同一时间只支持一种材质，因此是否开启双面灯光只是决定了背面是否完全照亮。

这里定义了两个聚光灯。用于 light1 的代码与用于 light0 的代码是相似的，除了 light0 有一个黄色（相当于混合红色和绿色）而 light1 有一个青色（相当于混合绿色和蓝色）。

```
//Light 0 is a spot light
self.baseEffect.light0.enabled = GL_TRUE;
self.baseEffect.light0.spotExponent = 20.0f;
self.baseEffect.light0.spotCutoff = 30.0f;
self.baseEffect.light0.diffuseColor = GLKVector4Make(
   1.0f, // Red
   1.0f, // Green
   0.0f, // Blue
   1.0f);// Alpha
self.baseEffect.light0.specularColor = GLKVector4Make(
   0.0f, // Red
   0.0f, // Green
   0.0f, // Blue
   1.0f);// Alpha
```

第三个灯光（light2）会被设置为一个简单的定向光。回顾一下第 4 章，定向光的位置是最后一个元素设置为 0.0 的一个四元素矢量。变量 spotLight2Position 会被初始化为矢量 {0.0f，0.5f，1.0f，0.0f}。这个位置表示这个定向光照射自一个无限远的光源。

```
// Light 2 is directional
self.baseEffect.light2.enabled = GL_TRUE;
self.baseEffect.light2Position = spotLight2Position;
self.baseEffect.light2.diffuseColor = GLKVector4Make(
   0.5f, // Red
   0.5f, // Green
   0.5f, // Blue
   1.0f);// Alpha
```

“-viewDidLoad”方法配置了聚光灯的大部分属性，然后这些属性就永远不会再改变了。但是，与定向光不同的是，聚光灯的位置和方向并不是预先指定的。按照惯例，OpenGL 会使用当前（实际上是在设置位置和方向时）的 model-view 矩阵来变换位置光源的位置和方向。因此，每当灯光坐标系变化时必须重新设置位置和方向。

例子 OpenGLES_Ch6_3 会旋转聚光灯来移动光束，同时光束会与波动变化的旗帜相互作用。OpenGLES_Ch6_3ViewController 使用了四个新属性来动画化光束的方向，它们是 spotLight0TiltAboutXAngleDeg、spotLight0TiltAboutZAngleDeg、spotLight1Tilt About-XAngleDeg 和 spotLight1TiltAboutZAngleDeg。与其他动画一样，基于时间的函数决定了属性值的变化。

```
- (void)updateSpotLightDirections
{
   // Tilt the spot lights using periodic functions for simple
   // smooth animation (constants are arbitrary and chosen for
   // visually interesting spot light directions)
   spotLight0TiltAboutXAngleDeg = -20.0f + 30.0f * sinf(
      self.timeSinceLastResume);
   spotLight0TiltAboutZAngleDeg = 30.0f * cosf(
      self.timeSinceLastResume);
   spotLight1TiltAboutXAngleDeg = 20.0f + 30.0f * cosf(
      self.timeSinceLastResume);
   spotLight1TiltAboutZAngleDeg = 30.0f * sinf(
      self.timeSinceLastResume);
}
```

OpenGLES_Ch6_3ViewController 的两个方法 -drawLight0 和 -drawLight1 都会变换基于聚光灯的倾斜角属性的 model-view 坐标系，设置一个聚光灯的位置和方向，然后绘制一个简单的“能够发光”的模型来标示灯光的位置和方向。下面的粗体代码会配置聚光灯为一个位于原点的位置光源，并且聚光方向总是向下指向 Y 轴的负方向。

“能够发光”的模型相应地使用一个接近原点的顶点网格，并且似乎指向 Y 轴的负方向。每次绘制一个灯光和模型时会创建一个本地坐标系，这个本地坐标系会产生在这个例子中看到的移动的灯光效果。

```
- (void)drawLight0
{
   // Save effect attributes that will be changed
   GLKMatrix4  savedModelviewMatrix =
      self.baseEffect.transform.modelviewMatrix;

   // Translate to the model's position
   self.baseEffect.transform.modelviewMatrix =
      GLKMatrix4Translate(savedModelviewMatrix,
          spotLight0Position.x,
```

```
            spotLight0Position.y,
            spotLight0Position.z);
    self.baseEffect.transform.modelviewMatrix =
       GLKMatrix4Rotate(
            self.baseEffect.transform.modelviewMatrix,
            GLKMathDegreesToRadians(self.spotLight0TiltAboutXAngleDeg),
            1,
            0,
            0);
    self.baseEffect.transform.modelviewMatrix =
       GLKMatrix4Rotate(
            self.baseEffect.transform.modelviewMatrix,
            GLKMathDegreesToRadians(self.spotLight0TiltAboutZAngleDeg),
            0,
            0,
            1);

    // Configure light in current coordinate system
    self.baseEffect.light0Position = GLKVector4Make(0, 0, 0, 1);
    self.baseEffect.light0SpotDirection = GLKVector3Make(0, -1, 0);

    [self.baseEffect prepareToDraw];
    [self.canLightModel draw];

    // Restore saved attributes
    self.baseEffect.transform.modelviewMatrix =
       savedModelviewMatrix;
}
```

6.4 动画化纹理

随着时间变换 model-view 坐标系会让渲染对象看起来像是相对于视点移动，反之亦然。随着时间改变纹理坐标系可以动画化几何体的纹理映射过程。示例 OpenGLES_Ch6_4 会加载并映射一个半透明纹理到例子 OpenGLES_Ch6_3 的飘扬旗帜上。这个例子会基于逝去的时间旋转纹理坐标系来产生图 6-8 中的屏幕截图。

图 6-8 映射到飘扬旗帜图形上的动画纹理

6.4.1 OpenGLES_Ch6_4 示例

OpenGLES_Ch6_4ViewController 的“-viewDidLoad”方法会利用下面的代码加载并配置几乎与例子 OpenGLES_Ch3_1 中的纹理相同的纹理。

```
// Setup texture0
CGImageRef imageRef0 =
   [[UIImage imageNamed:@"RadiusSelectionTool.png"] CGImage];

GLKTextureInfo *textureInfo0 = [GLKTextureLoader
   textureWithCGImage:imageRef0
   options:nil
   error:NULL];

self.baseEffect.texture2d0.name = textureInfo0.name;
self.baseEffect.texture2d0.target = textureInfo0.target;
```

添加到 OpenGLES_Ch6_4ViewController 的新方法“-updateTextureTransform”会旋转纹理坐标系。回顾一下第 3 章，纹理坐标系会分别沿着 S 轴和 T 轴从 0.0 跨度到 1.0。下面的代码会围绕纹理的中心旋转，具体过程是先平移到 {0.5，0.5}，然后旋转，最后再平移回 {–0.5，–0.5}。第 5 章已经讲到过纹理矩阵以及围绕一个点旋转的技巧。

```
- (void)updateTextureTransform
{
   // Rotate the texture
   // Translate to center of texture coordinate system
   self.baseEffect.textureMatrix2d0 =
      GLKMatrix4MakeTranslation(
         0.5,
         0.5,
         0.0);
   // Rotate
   self.baseEffect.textureMatrix2d0 =
      GLKMatrix4Rotate(
         self.baseEffect.textureMatrix2d0,
         self.timeSinceLastResume, // use interval as num radians
         0,
         0,
         1);
   // Translate back to texture coordinate system origin
   self.baseEffect.textureMatrix2d0 =
      GLKMatrix4Translate(
         self.baseEffect.textureMatrix2d0,
         -0.5,
         -0.5,
         0);
}
```

OpenGLES_Ch6_4ViewController 的“-update”方法会更新纹理矩阵以及聚光灯的方向，参见下面的代码。

```
////////////////////////////////////////////////////////////////
// This method is called automatically at the update rate of the
// receiver (default 30 Hz).
- (void)update
{
   [self updateSpotLightDirections];
   [self updateTextureTransform];
}
```

这些就是实现纹理旋转需要做的。飘扬旗帜的网格上的顶点位置的变化和纹理矩阵的变化结合起来产生了一个形象的复杂动画。在例子 OpenGLES_Ch6_4 中需要使用 AGLKTextureTransformBaseEffect 类，这是因为在写作本书时，GLKit 内置的 GLKBaseEffect 类还没有实现纹理矩阵。

6.4.2 OpenGLES_Ch6_5 示例

例子 OpenGLES_Ch6_5 会容纳在纹理贴图集（texture atlas）的单一 OpenGL ES 纹理中的多个 2D 图像。每个图像都与其他的图像稍有不同，并充当了短片中的一个帧。当这些帧快速连续地显示时，会呈现出运动的错觉。例子 OpenGLES_Ch6_5 使用了与 OpenGLES_Ch6_4 相同的基本技术来加载和映射一个纹理到几何图形上，只不过 OpenGLES_Ch6_5 使用了一个纹理贴图集。

OpenGLES_Ch6_5 的纹理贴图集图像有 1024 像素宽，1024 像素长。它被分为 8 行，每行 128 像素高；8 列，每列 128 像素宽，参见图 6-9。在这个纹理贴图集里有 64 个子图像空间，但在这个例子中只需要 51 个子图像空间。结果，在图 6-9 中的最后一行以及倒数第二行的一部分一直没有用到。

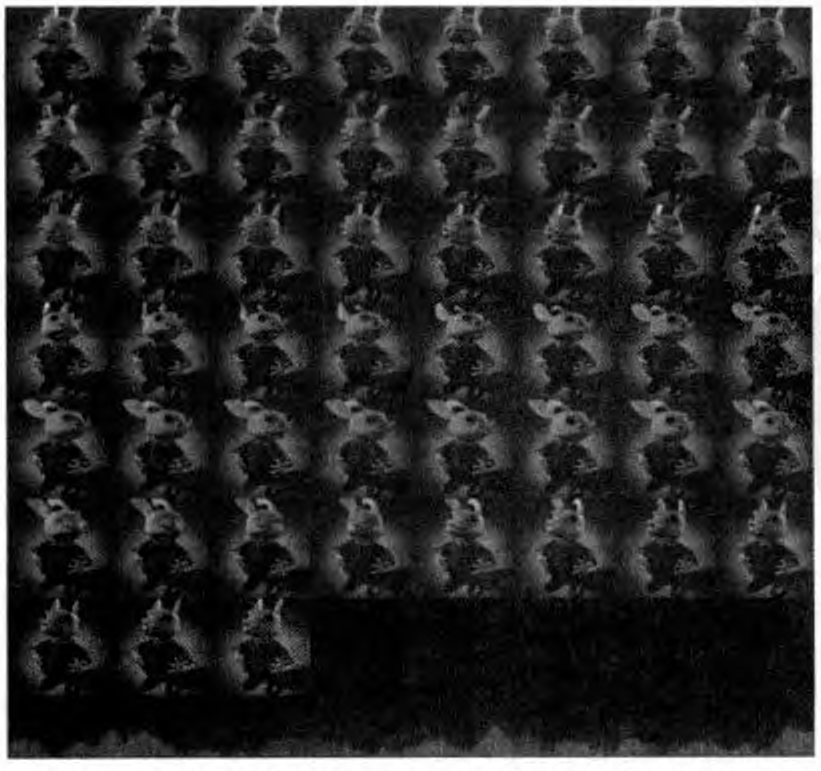


图 6-9　保存静态图像到一个网格的纹理贴图集

下面的常量用来辅助计算每帧动画在纹理贴图集内的位置。

```
//////////////////////////////////////////////////////////////////
// Constants used to calculate the texture position of each
// sub-image in a texture atlas that contains still frames from
// a movie
static const int numberOfMovieFrames = 51;
static const int numberOfMovieFramesPerRow = 8;
static const int numberOfMovieFramesPerColumn = 8;
static const int numberOfFramesPerSecond = 15;
```

和例子 OpenGLES_Ch6_4 一样，例子 OpenGLES_Ch6_5 会分配在 0.0 到 1.0 范围内的 S 和 T 纹理坐标到网格顶点之间。不用对纹理矩阵做任何修改，整个贴图集会被映射到网格几何图形上。但是，这个例子每次只需要渲染一个动画帧覆盖的网格。下面的代码首先平移纹理坐标系，以使原点 {0.0，0.0} 对应于当前动画帧的左下角，然后缩放纹理坐标系以使当前帧能够覆盖网格几何体：

```
- (void)updateTextureTransform
{
   // Calculate which sub-image of the texture atlas to use
   int      movieFrameNumber =
      (int)floor(self.timeSinceLastResume * numberOfFramesPerSecond) %
         numberOfMovieFrames;

   // Calculate the position of the current sub-image
   GLfloat   currentRowPosition =
      (movieFrameNumber % numberOfMovieFramesPerRow) *
      1.0f / numberOfMovieFramesPerRow;
   GLfloat   currentColumnPosition =
      (movieFrameNumber / numberOfMovieFramesPerColumn) *
      1.0f / numberOfMovieFramesPerColumn;

   // Translate to origin of current frame
   self.baseEffect.textureMatrix2d0 =
      GLKMatrix4MakeTranslation(
         currentRowPosition,
         currentColumnPosition,
         0.0f);
   // Scale to make current frame fill coordinate space
   self.baseEffect.textureMatrix2d0 =
      GLKMatrix4Scale(
          self.baseEffect.textureMatrix2d0,
          1.0f/numberOfMovieFramesPerRow,
          1.0f/numberOfMovieFramesPerColumn,
          1.0f);
}
```

运行例子 OpenGLES_Ch6_5 来观察在一个应用中动画的顶点数据、动画的灯光和动画的纹理结合起来的效果。同样，试着随着时间改变例子的视点。例如，使用类似 sinf() 的一个过滤器或者一个周期性函数来改变 GLKMatrix4MakeLookAt() 函数的参数。向例子 OpenGLES_Ch6_1 添加聚光灯来获得一个更加逼真的碰碰车模拟效果。

6.5 小结

除了 X、Y 和 Z 坐标系的三个维度之外，动画提供了第四个维度：时间。在本章中介绍的动画方法是多种类型的应用的基础，包括游戏、科学可视化和 Cocoa Touch 用户界面效果等。几乎所有的动画都是通过随着时间变换坐标系来实现的，不管是在一个场景内来回移动一个对象，还是变化聚光灯的方向，或是旋转纹理，又或者以第一人称视点在一个地牢迷宫中来回走动，都是这样。改变顶点属性会产生与本章例子中的飘扬旗帜相似的动画，但是为了获得最佳性能，应该避免频繁地修改顶点数据。

第 7 章和第 8 章会扩展在本章中介绍的 SceneModel 和 SceneAnimateMesh 类。对于大部分 3D 应用来说模型是必不可少的，这是因为非主流的、硬编码的顶点数据会很快变得难以控制。第 7 章会探索比本章中用到的碰碰车、溜冰场和“能够发光”的模型更加复杂的模型。第 8 章会演示使用尽可能少的运行时处理，来产生逼真的可视效果的多个技术。

第 7 章　加载和使用模型

截止到现在的所有例子使用的几何顶点属性都是编译到应用中的。第一个例子，OpenGLES_Ch2_1，硬编码了三个顶点的位置来定义一个三角形。第 5 章生成的 C 语言头文件也包含更复杂的几何定义。把顶点数据直接编译到应用中有利于快速加载，但是也有局限。

在运行时加载顶点属性和其他数据要比把数据直接编译到应用中更加灵活和动态。本章会介绍运行时数据加载。存在许多用于创建数据的工具，也存在许多互不兼容的存储数据的格式。因此想要解析文件中的数据并把加载的数据最终转换为顶点属性、纹理，以及 OpenGL ES 需要的效果的话，就可能需要使用多个技术。

在开始编码之前，有必要了解一下在运行时加载数据的原因。专注于最终目标有助于从无数的可用选项中选择一个技术方法。下面是假设的目标：

- 有一个可以接受的加载和运行性能。
- 减少运行时对稀缺内存的消耗量。
- 不必编译就可以向应用更新或者添加数据。
- 利用多个团队成员来简化应用的开发。

在应用加载时从 iOS 文件系统中读取数据永远不如把数据编译到应用中速度快。不过，如果小心使用的话，从文件系统读取数据也可以获得一个可接受的性能。关键是要以需要最少处理的方式来保存数据，以利于加载。另一种方法是延时加载数据，尽可能在最后一刻加载数据，或者在一个独立于主应用用户界面的线程中异步加载数据。最主要的目的是让应用尽可能快地响应用户。

编译到应用中的数据会在应用整个运行过程中一直寄存于 CPU 控制的内存中。为了渲染，数据（至少其中一些）的第二份副本必须要保存在 GPU 控制的内存中。在使用小的数据集时，保持相同数据的多个副本可能不是问题，但是嵌入式系统的内存是稀缺的。当有多个复杂场景时，保存两个副本所要耗费的额外存储空间应该是不可接受的。更糟糕的是，当同一时间不管怎样都只能看到一个场景时，在 CPU 控制的内存中同时保存多个场景就是浪费。只加载当前需要的数据，把它传递给 GPU 控制的内存，然后释放 CPU 控制的内存，这样可以极大地减少总内存需求。

打算与用户创建的数据交互的应用显然不能把数据编译到应用自身中。一些应用在运行时通过互联网接口下载数据。一个游戏可能会首先发布限制版，然后通过因特网捐助或者通过应用内购买来获得额外的关卡。理论上，运行时加载可以让所有这些情景成为可能。

最后，运行时加载通常可以简化应用的开发。在大型开发团队中，当数据生产与软件开发的进度不同时，运行时加载尤其重要。设计师希望在应用中能够及时地看到他们的创作，并做一些处理来实现快速迭代改进。加载设计师新数据的理想方式是不用重启应用就可以加载，更别说重新编译了。

7.1 建模工具和格式

用文本编辑器或是用代码来创建顶点属性数据是困难的，容易出错的，并且是枯燥乏味的。存在很多创建 3D 数据的建模工具，从顶点属性效果到动画效果、物理效果、灯光效果等。建模（modeling）是 3D 内容创建者用来描述 3D 数据模型开发过程的术语。表 7-1 对比了流行的建模工具。

表 7-1 流行 3D 建模工具比较

工具	网址	优点	缺点
Autodesk 3ds Max Products	usa.autodesk.com/3ds-max	动画人物模型，低多边形模型，材质管理；支持第三方插件	不支持 Mac OS X；昂贵；只提供免费试用版。
Autodesk Maya	usa.autodesk.com/maya	支持带有高级细节和灯光的电影级场景；支持 Mac OS X	昂贵；只提供免费试用版
Blender	www.blender.org	免费，开源，拥有强大的用户社区	用户界面复杂，并没有针对任何平台做界面优化
Google SketchUp	sketchup.google.com	免费基本版就能满足大多数需求；专业版相对便宜；界面简单优雅	动画能力不足；用于特殊效果的高级建模工具匮乏

表 7-1 列出了可用工具的一个很小的子集。未列出的工具可能是用于特殊需求的。维基百科上有一个更全面的对比，网址为：http://en.wikipedia.org/wiki/Comparison_of_3D_computer_graphics_software。

本书的模型是用 SketchUp 或者 Blender 制作的，有时是两个工具的结合。因为本书的例子需要面向尽可能多的读者；而 SketchUp 和 Blender 都是免费的，并且都支持 Mac OS X。

所有的 3D 建模工具都会用不同的并且常常是专有的文件格式来保存创建的模型数据。大部分工具可以从其他工具导入或者导出模型数据。有时，第三方插件会实现模型的导入与导出。几十年来，一直迫切需要一个用于工具间数据交换的能够可靠记录 3D 数据的文件格式。Wavefront 公司的 .OBJ 格式曾经是数据交换的主流格式，因为它容易

解析，但是它非常局限。这种格式不能保存现代应用所产生的关键信息。使用“.OBJ”格式在工具间交换模型会带来无意中丢失有用信息的危险。

今天的解决方案是 Khronos Group 所控制的 COLLADA 格式，OpenGL 标准也是由这个组织控制的。COLLADA 使用可扩展标记语言（XML）来保存模型数据，换句话说，就是以普通的文本文件来保存。它是免费使用的，没有特许权使用费或者其他费用，详细说明参见网址：www.khronos.org/collada。几乎所有的现代建模工具都使用 COLLADA 格式来导出模型，导入格式也是这样。

COLLADA 会保存几何体、材质、纹理、灯光、动画等到文件中，使用的是“.dae”扩展名。顶点数据通常是以一个紧凑的实例化的表示形式来组织的，这意味着相同的顶点数据可能会用在多个模型或者网格中。第 6 章介绍过网格，本章会详细说明这个概念。

COLLADA 实现了建模工具和 3D 应用之间的信息交换。通常，这个格式是无损传输数据的。但是 COLLADA 并不适用于保存 iOS 应用直接加载的数据。实现工具间的互通所需要的复杂性让这个格式难以解析。最好的做法是首先使用一个离线工具来读取 COLLADA 文件，然后用这个工具输出特别适合 iOS 应用的数据的一个子集。

资源管道

资源管道（asset pipeline）描述了一个工具序列，包括用于创建 3D 数据的工具、用于转换数据格式的工具、优化工具、问题测试工具，以及输出适用于在一个应用中高效加载的数据的工具。本书使用一个简单的资源管道，这个资源管道开始于 SketchUp 或者 Blender，参见图 7-1。

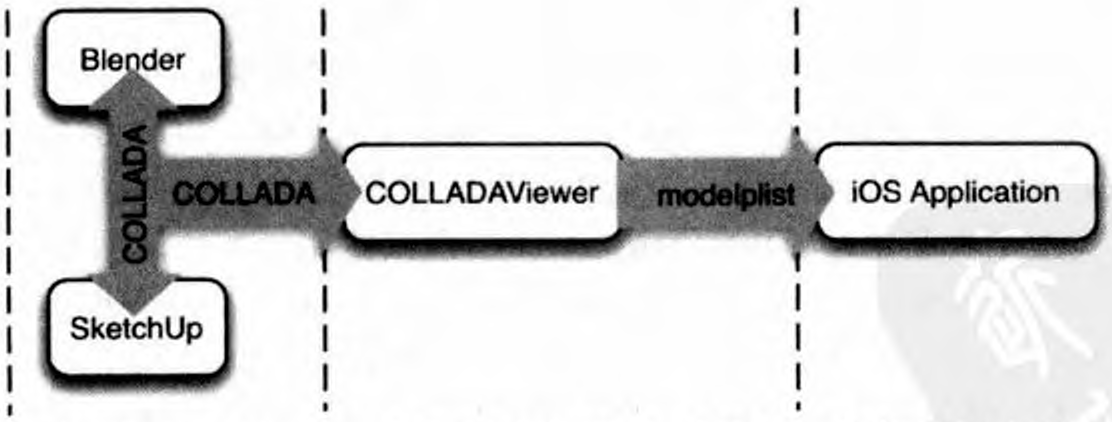


图 7-1　基于 COLLADAViewer 的资源管道

COLLADAViewer 应用运行在 Mac OS X 上，可以在本章的例子中找到，下载地址为 http://opengles.cosmicthump.com/learning-opengl-es-sample-code/。COLLADAViewer.app 会利用 Mac OS X 的 NSXMLDocument 和相关的 NSXML 类来解析“.dae”文件。使用比较少的 Objective-C 代码（只有五个类）就可以从 NSXMLDocument 中提取想要的原始数据并转换为网格和模型。

还存在能够解析“.dae”文件的其他工具。OpenCOLLADA 开源软件开发工具包（http://opencollada.org/）可以解析“.dae”文件，并为解析数据或者转换为其他格式提供了一个基础。很多建模工具使用 OpenCOLLADA 来实现导入和导出。已经使用 OpenCOLLADA 建立了 COLLADAViewer 应用的很多版本，不过坦率地说，OpenCOLLADA 是庞大的，并且有很多在 Mac OS X 上不常见的附属库。粗略估计，OpenCOLLADA 实现了数千个 C++ 类，这些类分散在 1200 个不同的源文件中。如果类似 C++“boost”这样的常见附属库也计算在内的话，建立 Max OS X 版 OpenCOLLADA 就需要编译由 420 万行代码组成的 2 万个 C++ 文件。对于 COLLADAViewer 来说，使用 Mac OS X 的内建库和 Objective-C 更合适。

解析“.dae”文件的其他选项包括官方但是过时的 COLLADA Document Object Model（DOM），网址为 http://sourceforge.net/projects/collada-dom/，以及 FCollada 商业类库。Open Asset Import 库（最有趣的项目之一），网址为：http://assimp.sourceforge.net/，支持包括 COLLADA 在内的很多 3D 模型格式。新项目应该考虑使用 Open Asset Import 库，部分原因是它会加载模型到一个简单的数据结构中，并且支持用于常见优化的各种后期处理步骤。

COLLADAViewer.app 足以把 COLLADA 文件转换为容易导入 iOS 应用中的数据，但是 COLLADAVewer 有一长串的限制。例如，COLLADAViewer 会假设每个“.dae”文件只包含一个模型。不支持任何的 COLLADA 库 / 实例对象。Blender 会自动导出带有适当网格的模型，但是在使用 SketchUp 时，在你导出 COLLADA 格式文件之前，你必须要“分解”所有的“组成部分”。在 SketchUp 中在模型上点击鼠标右键可以看到一个“Explode”上下文菜单。COLLADAViewer 一次只使用一个纹理图片；因此你必须要把所需的所有纹理图片结合到一个所有模型共享的大的纹理贴图集中。最后导入的模型的纹理图像会替代 COLLADAViewer 中的所有其他纹理图像。最好的做法是在映射到任何模型之前先使用各种建模工具创建纹理贴图集。当所有模型都使用相同的纹理图像时，它们导入 COLLADAViewer 的顺序就不再重要了。最后，COLLADAViewer 会假设模型的位置在坐标系原点附近。

注意 本书例子中的模型都是以米为单位制作的，但是 SketchUp 总是转换用户指定的单位为等量的英寸，最后导出以英寸为单位的 COLLADA 文件。无论在建模工具中使用的是什么单位和方向，COLLADAViewer 都会自动地将模型的方向转换

为“Y 轴向上”，并把模型缩放为以米为单位。在 COLLADAViewer 中很难看清尺寸大于几十米或者小于几厘米的模型。同样的，离坐标系原点的距离大于几米的模型在 COLLADAViewer 中也是看不到的。

图 7-2 显示的是导入并合并了三个不同的 COLLADA 文件后的 COLLADAViewer 应用的屏幕截图。使用鼠标滚轮或者在触摸板上用两个手指拖动来放大和缩小。复选框可以切换参考用的法向量和 –Z 坐标轴的显示。例子 OpenGLES_Ch7_1 重用了第 6 章的模拟碰碰车，图 7-2 中显示的是加载的模型。

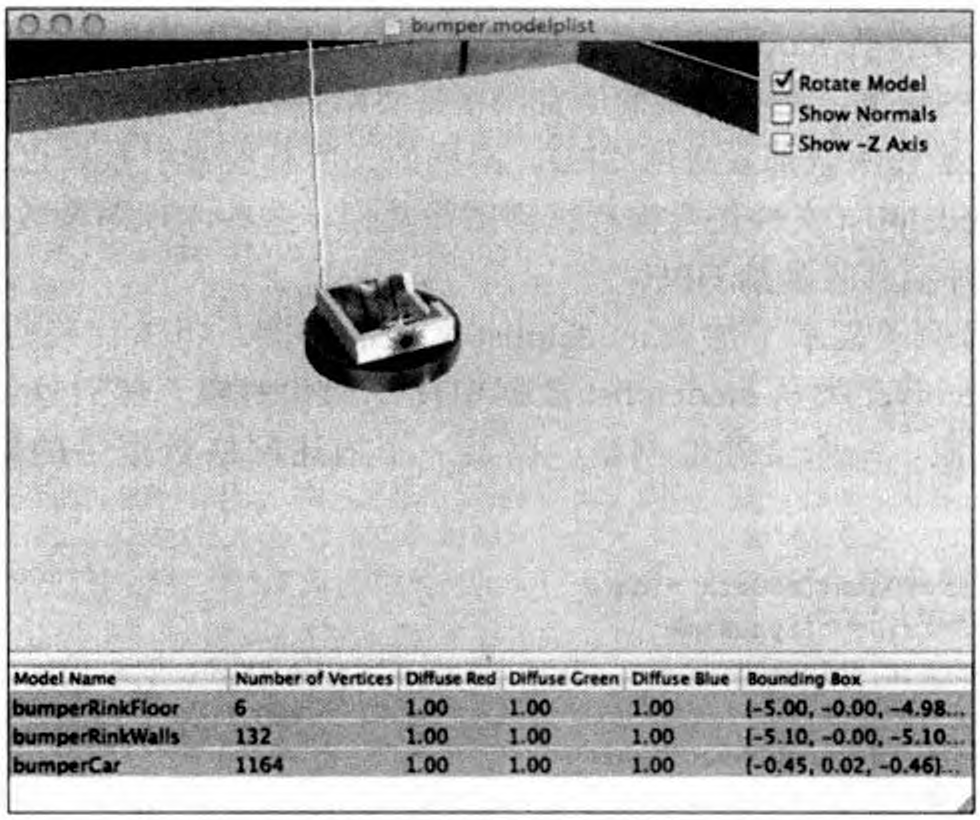


图 7-2　COLLADAViewer 的屏幕截图

即使有这些局限性，COLLADAViewer.app 仍然为创建一个综合的 Mac OS X 版 COLLADA 模型浏览器提供了一个起点。另一个例子是 XRay，其网址为 http://www.hfink.eu/collada/xray，使用了 OpenCOLLADA，提供一个 Mac OS X 平台的 Xcode 工程源码。

COLLADAViewer 会使用“.modelplist”扩展名的文件来保存导入的模型。属性列表（plists）定义了一个 Max OS X 和 iOS 原生的文件格式，详情请查询网址：http://developer.apple.com/library/ios/documentation/Cocoa/Conceptual/PropertyLists。属性列表可以把数据组织到一个结构化的仍然非常有效率的格式中。在 Mac OS X 和 iOS 上，应用常常要用到属性列表。例如，系统和应用的用户默认值都是以 plist 格式的文件保存的。简而言之，plist 文件可以保存序列化的 Cocoa 或者 Cocoa Touch 的 NSData、NSArray、NSDictionary、NSString、NSDate 和 NSNumber 对象的层次结构。plist 文件兼容 Mac

OS X 和 iOS，因此 COLLADAViewer.app 在 Mac OS X 上创建的 modelplist 文件可以直接被 iOS 应用读取（以紧凑的二进制文件的形式）。下一节会讲解在 modelplist 文件中使用的 Cocoa Touch 对象的具体层次结构。

7.2 读取 modelplist 文件

无论模型文件的格式是什么，GPU 最终处理的是 GPU 控制内存中的顶点属性缓存和索引缓存。用正确的格式有效地向 GPU 发送数据才能获得良好的应用性能。结合和转换 COLLADA 文件为 modelplist 文件的主要目的，仍然是可以被 iOS 应用快速加载。为了加快运行时的速度，所有模型的所有网格都要结合在一个单独顶点属性缓存中，然后就可以在一次操作中全部发送给 GPU。同样的，所有的索引数据也都存储在一个单独的缓存中。modelplist 文件还会保存纹理图像数据，这些纹理图像数据会在一个单独的纹理缓存操作中直接发送给 GPU。

下面的代码例子显示了加载 modelplist 文件的步骤。粗体代码行用 Cocoa Touch 的 NSKeyedUnarchiver 类从 modelplist 数据中反序列化对象。然后这段代码提取了一个 textureInfo 对象、一个合并的网格，以及一个描述网格数据中的模型名字的字典（dictionary）。

```
- (BOOL)readFromData:(NSData *)data
   ofType:(NSString *)typeName
   error:(NSError **)outError
{
   NSDictionary *documentDictionary =
      [NSKeyedUnarchiver unarchiveObjectWithData:data];

   self.textureInfo = [GLKTextureInfo
      textureInfoFromUtilityPlistRepresentation:
         [documentDictionary objectForKey:
         UtilityModelManagerTextureImageInfo]];

   self.consolidatedMesh = [[UtilityMesh alloc]
      initWithPlistRepresentation:[documentDictionary
      objectForKey:UtilityModelManagerMesh]];

   self.modelsDictionary =
      [self modelsFromPlistRepresentation:[documentDictionary
      objectForKey:UtilityModelManagerModels]
      mesh:self.consolidatedMesh];

   return YES;
}
```

plists 文件可能包含 Cocoa Touch 支持的对象的任意组合。plists 文件常常用来保存数组字典的字典以及类似的嵌套结构。modelplist 格式会存储一个特定的结构，但是 plists 文件天生是可扩展的。当所需元素存在时，modelplist 文件中可能要保存额外的数据。

注意　苹果公司规定了常规 plist 格式的标准。在 modelplist 文件中的特定对象结构支持本书中的例子。不同的应用肯定会保存不同的信息或者完全不用 plist 格式。一个应用使用哪种格式来加载模型数据都可以，只要这个格式快速、紧凑，并且支持你使用的资源管道。

modelplist 文件必须要包含特定的对象。首先，根对象是一个 NSDictionary。根对象是 NSKeyedUnarchiver 的“-unarchiveObjectWithData：”方法返回的对象。根对象包含了 plist 文件中的所有其他对象。字典保存了一个键值对的集合，并且为键快速查询做了优化。在代码中创建的字典可以使用几乎任意的对象作为键，但是在 plists 文件中的字典通常使用字符串作为键。modelplist 文件的根字典必须至少要包含三个键，即 textureImageInfo、models 和 mesh。

```
textureImageInfo, models, and mesh.
//////////////////////////////////////////////////////////////////
// Constants used to access model properties from a plist
// dictionary.
NSString *const UtilityModelManagerTextureImageInfo =
   @"textureImageInfo";
NSString *const UtilityModelManagerMesh =
   @"mesh";
NSString *const UtilityModelManagerModels =
   @"models";
```

在 modelplist 文件中，与每个根目录键相伴的对象也是字典。图 7-3 图解了这个格式。灰色显示的对象在每个 modelplist 文件中都是一样的。白色显示的对象包含要从文件中提取的独一无二的数据。在“models”字典里的键是所包含的模型的名字。每个模型必须要有一个独一无二的名字，因为每个键只能在一个字典中出现一次。

例子 OpenGLES_Ch7_1 提供了 UtilityTextureInfo、UtilityModelManager、UtilityModel 和 UtilityMesh 类，每个类都会从 modelplist 文件中提取相关的信息。UtilityTextureInfo 使用“textureImageInfo”字典。UtilityModelManager 使用“models”字典，同时创建一个 UtilityModel 实例的新字典，并且每个实例都会读取一个 <model name> 字典。UtilityMesh 类会从“mesh”字典中抓出数据。

接下来的一节包含了使用 UtilityTextureInfo、UtilityModelManager、UtilityModel 和 UtilityMesh 类的代码。在第 6 章中，从例子 OpenGLES_Ch6_2 到 OpenGLES_Ch6_5，网格和模型一直以大致相同的方式工作。网格就是共边三角形组成的网，并且网格定义了 3D 图形。网格数据的最初来源（硬编码或者从文件加载）并不重要。GPU

擅长使用类似索引顶点的优化来渲染网格。高效地向 GPU 发送静态 / 不变的网格数据实现了高性能渲染。

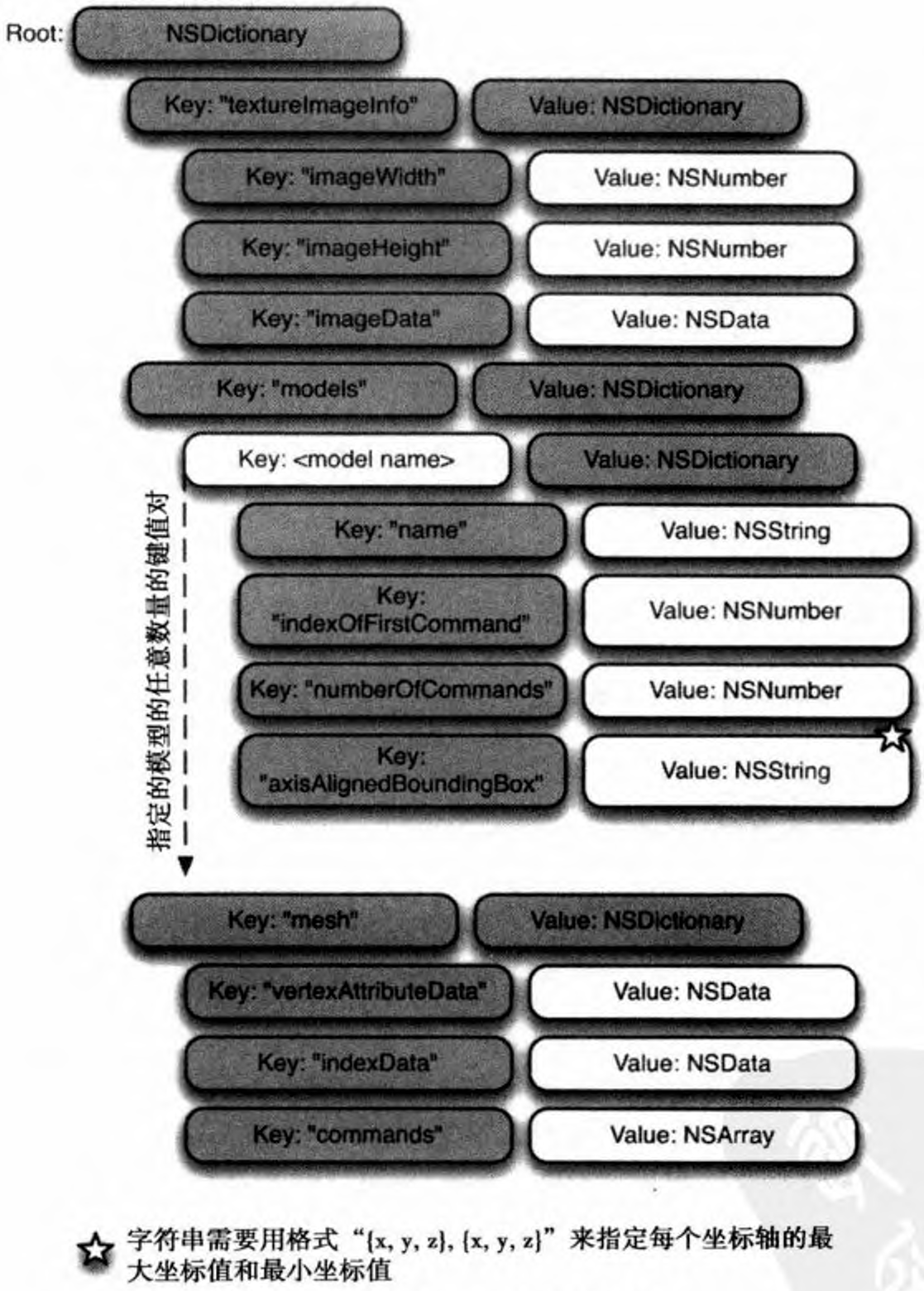


图 7-3　modelplist 格式的结构

7.3　OpenGLES_Ch7_1 示例

例子 OpenGLES_Ch7_1 在运行时加载了一个 modelplist 文件。这个 modelplist 文

件包含了一个网格，这个网格带有这个例子需要的所有模型的所有顶点属性和索引，同时还包含了一个所有模型共用的纹理图片。图 7-4 显示了例子 OpenGLES_Ch7_1 的输出。

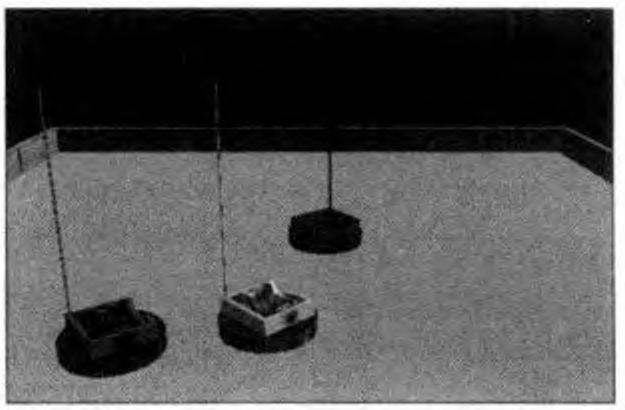


图 7-4　例子 OpenGLES_Ch7_1 的输出

从概念上说，模型是由一个或者多个网格组成的。模型可能会使用不同的坐标系变换和材质属性来绘制每个网格组件。设想一个人物模型的头部转动，但是肩部不转。那么头部和肩部应该是同一个模型内的两个不同的网格。

但是，使用 OpenGL ES 进行绘图最有效的方式是使用驻存在 GPU 控制的内存中的一个不变的大网格。一个比较好的折中方法是，通过指定模型使用部分所对应的顶点索引的范围，来为每个模型绘制一个单独的大网格中的不同子集。这正是 UtilityModel 和 UtilityMesh 类在 OpenGLES_Ch7_1 中的交互方式。所有的 UtilityModel 实例共享同一个 UtilityMesh 实例。模型会引用由网格保存的一个绘图命令数组。不同的命令使用不同的 OpenGL ES 模型绘制不同的网格子集，例如 GL_TRIANGLES 或者 GL_TRIANGLE_STRIP。三角形带是在第 6 章讲解的。

在 OpenGLES_Ch7_1 中的模拟碰碰车使用与 OpenGLES_Ch6_1 相同的基本模拟逻辑。两个例子中的 SceneCar 类几乎是相同的。在第 6 章的例子使用编译到应用中的模型数据的地方，OpenGLES_Ch7_1 添加了 UtilityModelManager 类来在运行时加载模型数据。

```
self.modelManager =
   [[UtilityModelManager alloc] initWithModelPath:modelsPath];
```

UtilityModelManager 类的“initWithModelPath:”方法使用指定路径中的文件的内容初始化了一个 NSDataobject 实例，接着使用在 7.2 节讲到的“-readFrom-Data:ofType:error:”方法来从数据中提取模型、网格和纹理对象。

```
NSData *data = [NSData dataWithContentsOfFile:aPath
   options:0
   error:&modelLoadingError];

if(nil != data)
{
   [self readFromData:data
      ofType:[aPath pathExtension]
      error:&modelLoadingError];
}
```

例子 OpenGLES_Ch7_1 中的 UtilityMesh 类是第 6 章的 SceneMesh 类的一个更通用版本。UtilityMesh 添加了一个用于保存命令数组的属性。UtilityModel 类有用来确定

适用于每个模型的网格绘图命令的范围的属性。

```
@property (assign, nonatomic, readwrite) NSUInteger
   indexOfFirstCommand;
@property (assign, nonatomic, readwrite) NSUInteger
   numberOfCommands;
@property (assign, nonatomic, readonly)
   AGLKAxisAllignedBoundingBox axisAlignedBoundingBox;
```

UtilityModels 还有一个由 COLLADAViewer 预计算的 axisAlignedBoundingBox 属性，并且会与模型一起保存在 modelplist 文件中。一个坐标轴对齐的边界框在空间上包含了模型的每个顶点。它会保存在模型网格中发现的最大和最小 X、Y、Z 值。坐标轴对齐的边界框定义了一个模型所占用的 3D 空间的体积，并且方便于多种用途。在本例和第 6 章的 OpenGLES_Ch6_1 例子中的 SceneCar 类都是用轴对齐的边界框属性来检测碰碰车之间的碰撞。

UtilityModel 的“-draw”方法会向网格发送一个消息，来让其使用模型命令范围内的命令绘图。

```
- (void)draw
{
   [self.mesh drawCommandsInRange:NSMakeRange(
      indexOfFirstCommand_, numberOfCommands_)];
}
```

UtilityMesh 的“-drawCommandsInRange:”方法会使用模型命令范围内的命令循环调用 glDrawElements() 函数，参见下面的代码。

```
- (void)drawCommandsInRange:(NSRange)aRange;
{
   if(0 < aRange.length)
   {
      const NSUInteger lastCommandIndex =
         (aRange.location + aRange.length) - 1;

      NSParameterAssert(aRange.location < [self.commands count]);
      NSParameterAssert(lastCommandIndex < [self.commands count]);

      for(NSUInteger i = aRange.location;
         i <= lastCommandIndex; i++)
      {
         NSDictionary *currentCommand =
            [self.commands objectAtIndex:i];
         const GLsizei  numberOfIndices = (GLsizei)[[currentCommand
            objectForKey:@"numberOfIndices"]
            unsignedIntegerValue];
         const GLsizei  firstIndex = (GLsizei){[currentCommand
```

```
            objectForKey:@"firstIndex"] unsignedIntegerValue];
        GLenum mode = (GLenum)[[currentCommand
            objectForKey:@"command"] unsignedIntegerValue];

        glDrawElements(mode,
            (GLsizei)numberOfIndices,
            GL_UNSIGNED_SHORT,
            ((GLushort *)NULL + firstIndex));
    }
  }
}
```

对比例子 OpenGLES_Ch7_1 和例子 OpenGLES_Ch6_1，归纳一下网格和模型所代表两种核心概念的不同实现方式。即使是两种实现方式，在两个工程中的网格和模型类的能力也是相当相似的，以至于 SceneCar 类可以同时使用能够相互切换的两种实现方式。最终置于 GPU 控制内存中的是相同的顶点属性，即使它们来自不同的源。这个概念让使用任意的工具来实现任意的资源管道成为可能，只要网格与模型之间存在这个关键关系。

背面剔除

例子 OpenGLES_Ch7_1 在 OpenGL ES 状态机中使用了一个新模式：GL_CULL_FACE。如在第 6 章提到的，每个三角形都有一个前面和一个后面，并且这个前后是由 OpenGL ES 接收三角形顶点的顺序决定的。剔除指的是一个可选的优化，是 OpenGL ES 用来在执行过多的处理之前及早丢弃顶点属性的。背面剔除是指丢弃那些背离观察者的三角形。当观察一个模拟的实心物体时，背面通常朝向物体内部，因此丢弃背面三角形并不会影响场景的渲染结果；即使没有剔除背面，也只有从一个实心物体的内部看时，才能看到背面。

OpenGLES_Ch7_1ViewController 开启 GL_CULL_FACE 并不只是为了优化；从 SketchUp 加载的模型通常是“双面”的。在 SketchUp 中，这个术语的意思是三角形的前面和后面有不同的材质、不同的颜色、不同的纹理。不设定背面剔除的话，OpenGL ES 就会试图把“双面”三角形的两面都渲染了，但是这个双面实际上是会产生 Z 冲突的共面图元。

第 5 章讲到过 Z 冲突。在例子 OpenGLES_Ch7_1 中，Z 冲突的意思是帧缓存的像素颜色渲染缓存中的最终片元的颜色是随机的，可能来自“双面”三角形的前面，也可能来自后面。实际上，每个面决定一半的片元，并且三角形会因为片元颜色来回翻转而闪烁。

背面剔除通过只渲染前面而避免了 Z 冲突。下面来自 OpenGLES_Ch7_1ViewController 的粗体代码开启了背面剔除。

```
- (void)glkView:(GLKView *)view drawInRect:(CGRect)rect
{
   // Clear back frame buffer (erase previous drawing)
   // and depth buffer
   [((AGLKContext *)view.context)
      clear:GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT];

   // Cull back faces: Important! many Sketchup models have back
   // faces that cause Z fighting if back faces are not culled.
   [((AGLKContext *)view.context) enable:GL_CULL_FACE];

   // Calculate the aspect ratio for the scene and setup a
   // perspective projection
   const GLfloat  aspectRatio =
      (GLfloat)view.drawableWidth / (GLfloat)view.drawableHeight;
  self.baseEffect.transform.projectionMatrix =
     GLKMatrix4MakePerspective(
        GLKMathDegreesToRadians(35.0f),// Standard field of view
        aspectRatio,
        4.0f,   // Don't make near plane too close
        20.0f); // Far arbitrarily far enough to contain scene

  [self.modelManager prepareToDraw];
  [self.baseEffect prepareToDraw];

   // Draw the rink
   [self.rinkModelFloor draw];
   [self.rinkModelWalls draw];

   // Draw the cars
   [cars makeObjectsPerformSelector:
      @selector(drawWithBaseEffect:)
      withObject:self.baseEffect];
}
```

7.4 高级模型

深入学习高级模型超出了本书的范围，但是可以介绍几个概念和术语，来为以后的学习打下基础。本节会讲解一些相当复杂的主题，这些主题通常是想要动画化人物模型或者动画化某些类型的网格模型的程序员感兴趣的。如果你急切地想进入下一个主题，那么可以跳过本节，但是本节有一些有趣的例子。

7.4.1 骨骼动画

骨骼动画（skeletal animation）描述的是一个模型的造型和动画技术。网格提供了

与前面例子相似的模型的可视部分，但是除此之外，还有一个不可见的用来变形网格的由“关节”和“骨头”相互连接组成的结构，这个结构又被称为骨骼（armature）或者骨骼绑定（rig）。活动骨骼时，网格顶点也会跟着活动。例如，一个代表人物的腿和脚的网格会以代表踝关节、膝关节和髋关节的骨骼关节为中心来抬起和摇摆。术语骨骼动画暗示了它的主要用途就是活动用来模拟人物或者其他动物的模型，但是这个方法也适用于几乎所有的网格动画，包括第 6 章的飘扬旗帜的效果。

当动画化使用复杂网格的模型时，骨骼动画可以提供一个重要的优化。向 GPU 发送相对于几个骨骼关节的新方向，要快于重新计算并发送一个网格中每个顶点的位置。如在本节和下一节所讲解的，每个骨骼关节都是由一个变换矩阵控制的。GPU 已经使用在第 5 章介绍的 model-view 矩阵和投影矩阵变换了每个顶点的位置。使用骨骼只为每个顶点增加了很少的额外计算。创建模型的设计师通常是通过直觉找出骨骼动画的。通过移动关节来摆一个手臂姿势要比手动地移动相应网格中的每个顶点容易很多。几乎所有支持动画的建模工具都使用骨骼动画。

骨骼关节由一个位置和一个变换矩阵组成。这个位置实际上是相对于一个可选的“父”关节的位移。从一个关节到下一个关节之间的位移代表了一个不可见的骨头。多个关节形成一个层次结构，使得父关节的变化可以影响所有与之相关的“子”关节。例如，当肘位置变化的时候，与之相连的腕也会移动，以保持肘和腕之间的距离不变。任意关节的位置都可以使用它父关节的变换矩阵与这个关节自己的变换矩阵的级联计算出来。下面的代码会返回一个关节的累积变换。这段代码使用了 5.5 节中的一个复合变换：

```
//////////////////////////////////////////////////////////////////
// Returns the cumulative matrix that includes parent transforms
- (GLKMatrix4)cumulativeTransforms;
{
   GLKMatrix4 result = GLKMatrix4Identity;

   if(nil != parent)
   {
      result = [parent cumulativeTransforms];
   }

   GLKVector3 d = self.cumulativeDisplacement;

   // Use the classic recipe for transform about a point:
   // translate to the location of the joint, rotate, and
   // translate back.
   result = GLKMatrix4Translate(result, d.x, d.y, d.z);
   result = GLKMatrix4Multiply(result, self.matrix);
   result = GLKMatrix4Translate(result, -d.x, -d.y, -d.z);

   return result;
}
```

例子 OpenGLES_Ch7_2 会帮助你可视化关节。从一个 modelplist 文件加载了三个简单模型。为了保持例子简单，模型的长度同时决定了关节之间的距离。在图 7-5 中显示的用户界面滑动条可以让你测试关节旋转时模型的效果。第一个关节，就是没有父关节的那个关节，位于最底部。

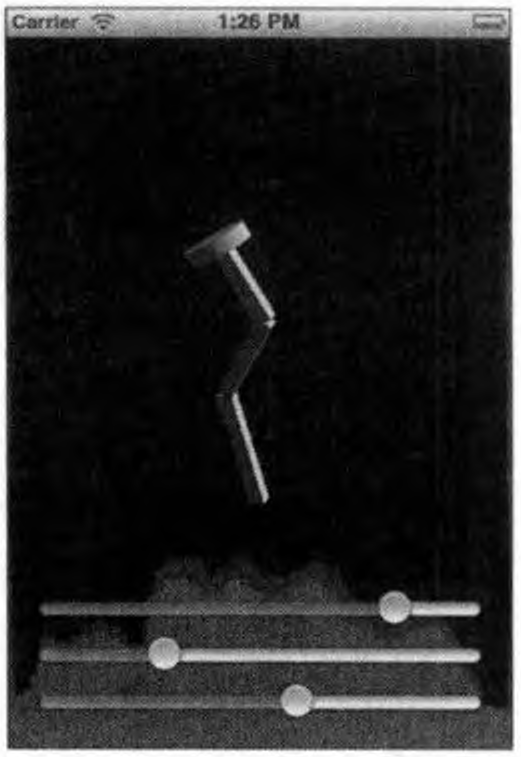


图 7-5　用来可视化骨骼的模型

例子 OpenGLES_Ch7_2 使用 GPU 来实现基于骨骼的动画。GLKBaseEffect 的一个新子类 UtilityArmature-BaseEffect 会创建并使用一个自定义的 OpenGL ES Shading Language 程序。使用 UtilityArmatureBaseEffect 的“-prepareToDrawArmature”方法来请求基于骨骼的动画。继承来的“-prepareToDraw”方法仍然能够工作，使用的是标准的 GLKit Shading Language 程序。UtilityArmatureBaseEffect 有一个 jointsArray 属性，用来保存 UtilityJoint 实例。在运行时更新关节属性，然后调用“-prepareToDrawArmature”方法来发送信息，这里的信息指的是 Shading Language 程序重新计算 GPU 当前内存中的顶点和法向量所需要的信息。

OpenGLES_Ch7_2ViewController 类包含的下面代码会创建三个关节并添加它们到例子的 UtilityArmatureBaseEffect 实例。关节位置是用加载模型的轴对齐边界框进行初始化的。关节位置和方向通常是直接从模型文件中加载来的，而不是计算出来的；本例使用这个尺寸的模型仅仅是为了方便和简单。

```
// Create collection of joints
UtilityJoint *bone0Joint = [[UtilityJoint alloc]
   initWithDisplacement:GLKVector3Make(0, 0, 0)
   parent:nil];
float bone0Length = self.bone0.axisAlignedBoundingBox.max.y -
   self.bone0.axisAlignedBoundingBox.min.y;

UtilityJoint *bone1Joint = [[UtilityJoint alloc]
   initWithDisplacement:GLKVector3Make(0, bone0Length, 0)
   parent:bone0Joint];
float bone1Length = self.bone1.axisAlignedBoundingBox.max.y -
   self.bone1.axisAlignedBoundingBox.min.y;

UtilityJoint *bone2Joint = [[UtilityJoint alloc]
   initWithDisplacement:GLKVector3Make(0, bone1Length, 0)
   parent:bone1Joint];

baseEffect.jointsArray = [NSArray arrayWithObjects:
   bone0Joint,
```

```
    bone1Joint,
    bone2Joint,
    nil];
```

每个模型都被赋给了一个单独的关节索引，使用与：[self.bone0 assignJoint:0] 相似的表达式。关节索引最终会与模型网格内的每个顶点的其他属性一起保存。Shading Language 程序会使用关节索引来查询每个顶点是由哪一个关节矩阵变换的。在 Shading Language 程序内，每个关节矩阵都已经与投影矩阵、model-view 矩阵和它的父关节的矩阵级联起来了，因此每个顶点（如这个关节自身、这个关节的父关节、这个关节的祖父关节等）都是使用层次结构中的所有关节的累加影响来变换的。

需要定义新的常量来添加额外属性。OpenGL ES 2.0 并没有为属性常量赋予任何内在意义。数据的解析取决于 Shading Language 程序。由苹果的 GLKBaseEffect 类生成的 Shading Language 程序会赋予常量含义，这些常量包括 GLKVertexAttribPosition、GLKVertexAttribNormal、GLKVertexAttribColor、GLKVertexAttribTexCoord0 和 GLKVertexAttribTexCoord1。除了 GLKVertexAttribColor 常量外，其他常量在前面的例子中都使用过。

```
typedef enum {
    UtilityArmatureVertexAttribPosition =
       GLKVertexAttribPosition,
    UtilityArmatureVertexAttribNormal =
       GLKVertexAttribNormal,
    UtilityArmatureVertexAttribTexCoord0 =
       GLKVertexAttribTexCoord0,
    UtilityArmatureVertexAttribTexCoord1 =
       GLKVertexAttribTexCoord1,
    UtilityArmatureVertexAttribJointMatrixIndices,
    UtilityArmatureVertexAttribJointNormalizedWeights,
} UtilityArmatureVertexAttrib;
```

UtilityArmatureBaseEffect 重定义了 GLKit 的属性并且添加了两个新的属性，UtilityArmatureVertexAttribJointMatrixIndices 和 UtilityArmatureVertexAttribJointNormalizedWeights。这个例子的 UtilityMesh 类会把关节信息保存在一个缓存中，并会以一个与其他每顶点属性相似的方式向 GPU 发送额外的值。

```
glEnableVertexAttribArray(
   UtilityArmatureVertexAttribJointMatrixIndices);
glVertexAttribPointer(
   UtilityArmatureVertexAttribJointMatrixIndices,
   4,                    // number of coordinates
   GL_FLOAT,             // data is floating point
   GL_FALSE,             // no fixed point scaling
   sizeof(UtilityMeshJointInfluence),// bytes per vert
   (GLubyte *)NULL +
      offsetof(UtilityMeshJointInfluence, jointIndices));
```

```
glEnableVertexAttribArray(
   UtilityArmatureVertexAttribJointNormalizedWeights);
glVertexAttribPointer(
   UtilityArmatureVertexAttribJointNormalizedWeights,
   4,                    // number of coordinates
   GL_FLOAT,             // data is floating point
   GL_FALSE,             // no fixed point scaling
   sizeof(UtilityMeshJointInfluence),// bytes per vert
   (GLubyte *)NULL +
      offsetof(UtilityMeshJointInfluence, jointWeights));
```

管理额外顶点属性的代码使用了与 UtilityMeshVertex 数据结构相似的数据结构 UtilityMeshJointInfluence。下一节会讲解 UtilityMeshJointInfluence 和关节权重。

OpenGLES_Ch7_2ViewController 类还包含移动滑动条时更新关节的代码。下面的方法接收滑动条返回的浮点值，并会使用这个值来旋转 jointsArray 中索引为 0 的关节。用滑动条的值乘以 M_PI 和 0.5 表示在正负 90 度的范围内调节第一个参数的弧度值。相似的方法更新了其他的两个关节。

```
- (void)setJoint0AngleRadians:(float)value
{
   joint0AngleRadians = value;

   GLKMatrix4  rotateZMatrix = GLKMatrix4MakeRotation(
      value * M_PI * 0.5, 0, 0, 1);

   [(UtilityJoint *)[baseEffect.jointsArray objectAtIndex:0]
      setMatrix:rotateZMatrix];
}
```

几乎可以为关节设置任意的变换。绕着 Z 轴旋转只是为了保持例子的简单性。试着改变代码，以绕着 X 轴旋转或者平移。缩放也可以正常工作，但是当缩放因数不均匀时（在一个方向上的缩放幅度与其他方向的缩放幅度不一致），Shading Language 程序内的光线方程式就会得出低质量的结果。

与第 5 章的一些例子中的 UtilityTextureTransformBaseEffect 类一样，相关例子包含了 UtilityArmatureBaseEffect 类的源代码以及它的 Shading Language 程序。UtilityArmatureBaseEffect 会从关节层次结构中提取出包含累积变换的关节矩阵，并把它们与 model-view 矩阵和投影矩阵级联。与此同时，它还为变换法向量计算新矩阵。UtilityArmatureBaseEffect 会向 GPU 发送一个结果矩阵的数组。每个顶点的 UtilityArmature-VertexAttribJointMatrixIndices 属性决定了每个顶点会受 UtilityArmatureBaseEffect 提供的数组中的那些矩阵的影响。

注意 法向量变换矩阵是 model-view 累积变换的逆矩阵的转置矩阵。如果你现在不理解的话，不用担心。这是一个很少需要直接注意的小细节。GLKBaseEffect 会自

动按需计算“法线矩阵”。类似的，UtilityArmatureBaseEffect 类会按需计算一个“法线矩阵”的数组。

图 7-6 展示了关节矩阵和每个顶点的 UtilityArmatureVertexAttribJointMatrixIndices 属性之间的关系。同一时间内 UtilityArmatureBaseEffect 可以支持多个关节矩阵，但是并不是每个矩阵都被每个顶点所引用。

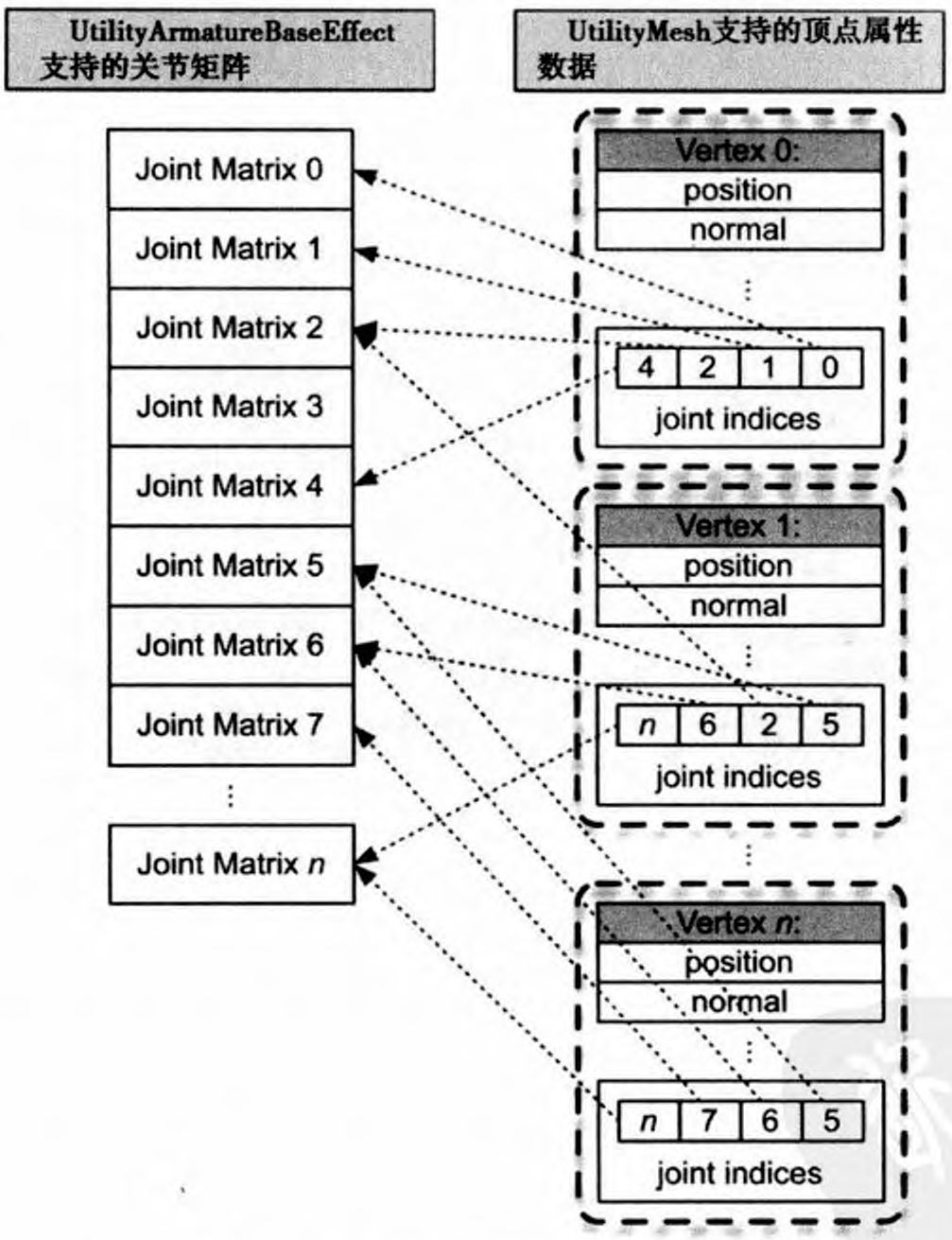


图 7-6　每顶点索引选择关节矩阵

7.4.2 蒙皮

如图 7-5 所示，当关节方向变化时，模型之间会出现缺口。在例子 OpenGLES_Ch7_2 中产生的这个结果是因为每个顶点只受一个关节的影响，并且不同的“骨头”是

用不同的模型绘制的。当一个单独的网格内的顶点直接受到多个关节的影响时，就有可能获得更有机、更平滑的网格动画。为了实现这一点，需要在每个顶点内保存多个关节索引。Shading Language 程序会查询每个顶点引用的所有关节。每个关节对于一个顶点的最终位置的影响取决与缩放因数，又叫做权重（weights）。一个关节可能有 50% 的影响力，另一个关节可能有 30% 的影响力，然后还有两个关节每个起 10% 的作用。这个比率并不重要，只要总影响力加起来等于 100%。

图 7-7 包含了例子 OpenGLES_Ch7_3 的两个屏幕截图，并且显示了多个关节联合影响的效果。设置每个顶点的关节索引和权重的过程就叫做蒙皮（skinning）。按比例混合多个关节对于每个网格顶点的影响力来变形网格，就好像在骨骼上拉伸皮肤。

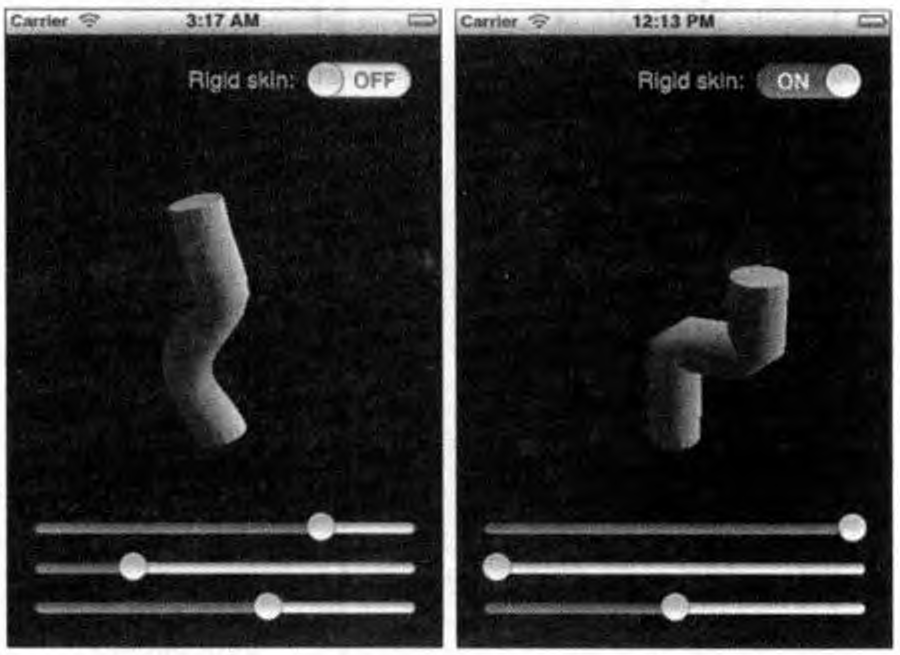


图 7-7　多个关节影响顶点位置的效果

OpenGLES_Ch7_3 使用了与 OpenGLES_Ch7_2 相同的关节，但是绘制了一个单独的叫做“tube”的大网格，而不是多个不同的骨头模型。把玩一下例子 OpenGLES_Ch7_3，感受一下移动滑动条时所产生的网格变形效果。

平滑的网格变形是通过多次变换每个网格顶点的位置来实现的：每个索引关节计算一次。如果得到的多个变换位置被求出平均值，那么计算出来的位置就相当于每个索引关节施加相等的影响力的蒙皮效果。权重提供了额外的控制力。使用权重后，一个关节可能有一个大的影响力，同时另一个关节却有一个非常小的影响力。

例子 OpenGLES_Ch7_3 中的“Rigid skin：”用户界面开关可以切换两种不同的“皮肤”。图 7-7 中的左图会按比例加重每个关节的影响力，这个比例的依据是这个关节与每个顶点之间的距离。距离远的关节比附近的关节影响小。当“Rigid Skin：”开关处于“开”状态时，会设置权重以让每个顶点只受到这个顶点下面离它最近的关节的影响，但是别忘了，每个关节已经包含了它的父关节对其的影响。

使用蒙皮可以创建类似带有皮肤的人物的复杂模型，并且这些皮肤会逼真地随着关

节的活动而变形。当膝盖弯曲时，这个膝盖附近的皮肤会拉伸，但是大腿附近的皮肤不会受影响。

注意　COLLADA 文件能够存储关节和权重。网页：http://www.wazim.com/Collada_Tutorial_1.htm 解释了怎么解析及使用 COLLADA 关节和权重。COLLADA 文件还存储了多个与关节方向相关的设置，这为骨骼动画又添加了另一维。方向的每个设置都类似于 OpenGLES_Ch7_3 中的滑动条的位置。每个设置都会创建一个与众不同的“姿势”。一个设置可能会让人物的手臂放在两侧，头看向肩膀。另一个设置可能会让人物的手臂抬起，头低下。在两个方向设置之间插值可以产生随着时间从一个姿势过渡到另一个姿势的平滑动画。

例子 OpenGLES_Ch7_3 在 UtilityModel 类中实现了自动指定关节和权重的两个不同的算法。详细的代码并不是很重要，因为自动蒙皮的模型在实际的应用中很少使用。相比在运行时实时计算关节的影响，在加载模型剩余部分的同时从一个文件加载关节索引和权重的方式要更好一点。设计师会用线下建模工具指定权重，以此来实现对于蒙皮的精巧控制。例子 OpenGLES_Ch7_3 的两个方法，“-automaticallySkinSmoothWithJoints:”和“–automaticallySkinRigidWithJoints:”，也要受到那个不利因素的影响，这个不利因素指的就是它们的工作同样需要访问网格顶点数据。理想情况下，在 CPU 控制内存中的网格顶点的副本会在被发送给 GPU 后尽快地被丢弃掉，但是为了支持重蒙皮的动态计算，这些网格数据就必须要保存在 CPU 控制的内存中。

例子 OpenGLES_Ch7_2 和 OpenGLES_Ch7_3 都会使用下面的数据结构来存储每个顶点的关节索引和权重：

```
//////////////////////////////////////////////////////////////////
// Type used to store vertex attributes
typedef struct
{
   GLKVector4 jointIndices; // encoded float for Shading Language
   GLKVector4 jointWeights; // weight factor for each joint index
}
UtilityMeshJointInfluence;
```

UtilityMeshJointInfluences 有一些与生俱来的古怪与限制。每个顶点只会保存四个关节索引和四个关节权重。四个关节通常就足以产生一个逼真的蒙皮效果。请记住，多个关节可能被用来变形一个单独的网格。这些关节中的任何一个都可能潜在地影响网格中的一个单独顶点。每个顶点选择四个关节。虽然索引值通常是整数或者字节，但是关节索引是用 GLKVector4 数据结构中的四个浮点值来存储的。OpenGL ES 2.0 Shading Language 不能接收每个顶点的整数或者字节属性。这是大部分现代嵌入式 GPU 的一个限制。OpenGL ES 2.0 Shading Language 会为将来的执行保留类型的名字，比如短整型

和字节。

7.4.3 逆动力学和物理模拟

古希腊术语动力学通常指的是物体的移动。现代背景下，动力学指的是使用骨骼动画让人物摆一个姿势所需要的关节方向变化的序列。正动力学定义了关节方向变化的一个序列：旋转肩膀 15 度，接着旋转肘 45 度，然后旋转胯 70 度，最后旋转手指关节来挠头。逆动力学提出了一个更有趣和前卫的方法。首先是目标，手指触摸头顶，然后返回来找出实现这个目的所需的所有关节方向变化。

使用 GPU 优化的逆动力学系统，可以最大限度地减少消息发送的数量。可以简单地发送末尾关节（或者假想的骨头）的最终位置，而不用发送每个关节的最新位置。所有其他关节的方向都可以用请求的最终位置推导出来。

逆动力学会变得非常复杂。为了正确工作，必须要遵守特定模型的约束条件。例如，手不允许从头部穿越达到头顶；肘不允许向后弯曲。

实时物理模拟，尤其是逆动力学是热点探讨主题。机器人公司、建模工具制造商、3D 模拟器开发商都在竞相提供通用解决方案。与此同时，流行的开源“Bullet Physics”引擎提供了很多作品。Bullet 会检测两个骨骼组件之间的碰撞，这至少会防止手臂穿过躯干或者肘向后弯曲。这仅仅揭开了 Bullet Physics 引擎能力的表面。它可以结合 Blender 和 SketchUp 并导出 COLLADA 文件。在网址：http://bulletphysics.org/ 中可以找到 Bullet Physics 以及它在 Mac OS X 平台上的 Xcode 演示工程。

7.5 小结

代表 3D 物体形状的模型是由一个或者多个网格定义的。在运行时加载模型为很多 3D 应用提供了重要的灵活性。通过脱机预处理数据并用原生格式保存数据，避免了嵌入式系统加载模型时所需的复杂解析和数据转换。模型可以非常复杂，但是它们是由简单的几何属性组成的。本章所使用的模型数据格式没有任何特别。利用模型和网格这两个概念可以实现使用任意工具和任意格式的任意资源管道。

模型骨骼、骨骼动画和蒙皮会使用 GPU 在运行时变形网格。通过更新相对少量的关节矩阵，避免了更新大量的顶点数据，也避免了向 GPU 重复发送数据。设计师通过设定关节和权重来实现对于骨骼动画和蒙皮效果的精巧控制。COLLADA 文件可以存储设计师提供的数据。骨骼还为物理模拟引擎检测碰撞提供了数据，包括模型与其他物体或者模型的一部分与自身的碰撞。

下一章会讲解特效。特效是用来增强渲染场景的外观的 3D 骗局和假象。特效可以让观察者在一个场景中看到更多实际不存在的细节。

第8章 特　效

本章会讲解增加 3D 应用表面细节的三个技术，并且不会明显地增加需要处理的顶点的数量。每个技术使用的都是 2D 图像，设计这些 2D 图像是为了欺骗眼睛的，并且即使是在一个 3D 场景内移动和摇摆时也分辨不出来。第一个是天空盒（skybox），就是把渲染场景包裹在一个六面都是纹理图片的立方体中。这些纹理图片会让几何体看起来就像是一直延伸到一个假想的地平线。第二个效果是粒子（particle），粒子可以高效地模拟爆炸、水流，或者浓烟滚滚的效果。最后是公告牌（billboard），公告牌可以利用对准视点的纹理三角形来掩饰深度的缺失。

8.1 天空盒

图 8-1 显示了一个没完全展开的天空盒，以及一个天空盒内部的场景渲染图。无论视点在天空盒内怎么移动，天空盒的纹理都会产生一个一直延伸到地平线的全景假象。

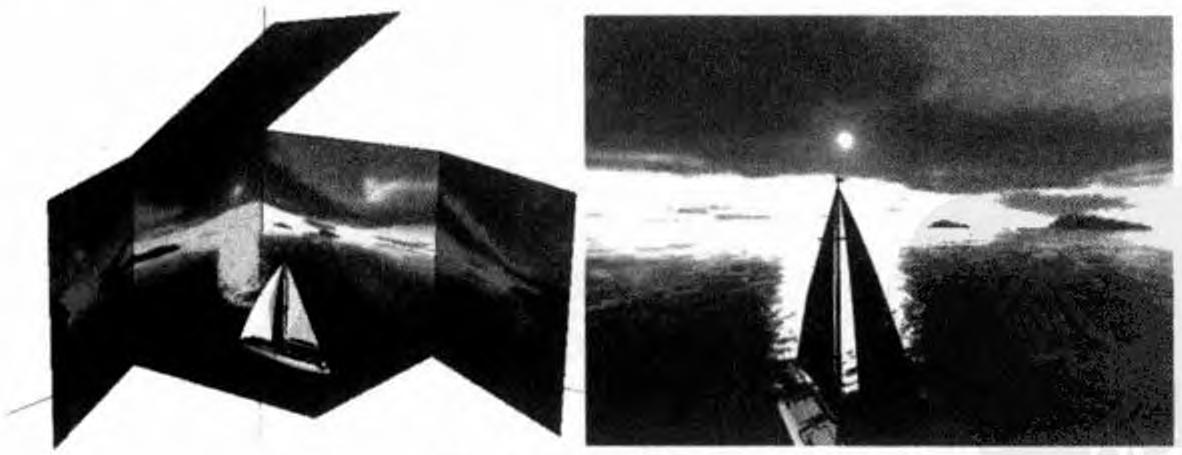


图 8-1　天空盒假象

天空盒只用了 12 个三角形，即立方体的六个面，每个面需要两个三角形。OpenGL ES 提供了一个叫做立方体贴图（cube mapping）的专门用以产生类似天空盒效果的纹理贴图模式。立方体的每个面都有一个 2D 纹理贴图。当渲染时，OpenGL ES 会基于一个矢量来计算每个片元的颜色，这个矢量是指从视点到立方体表面上的某个位置之间的

3D 方向矢量。立方体表面上的这个位置对应于六个面中的一个面的纹理内的 S 和 T 纹素坐标。第 3 章已经讲解了 S 和 T 纹素坐标。

图 8-2 显示的是放置在天空盒内的一个假想摄像头要渲染的体积。就图中显示的摄像头方向来说，一个从镜头的中心延伸到这个渲染体积内的任意一点的矢量会与三个面中的一面相交，北面、西面，或者底面。矢量与面的交点决定了要取样六个纹理中的哪一个。取样纹理的 S 和 T 坐标是由矢量与面的交点位置决定的。例如在图 8-2 中，指向渲染场景右上角的矢量大约会与北面的中心相交，这会让 S 和 T 坐标大约处于北面纹理的 {0.5，0.5} 位置处。与此相反，一个指向场景左下角的矢量会与底面纹理相交。

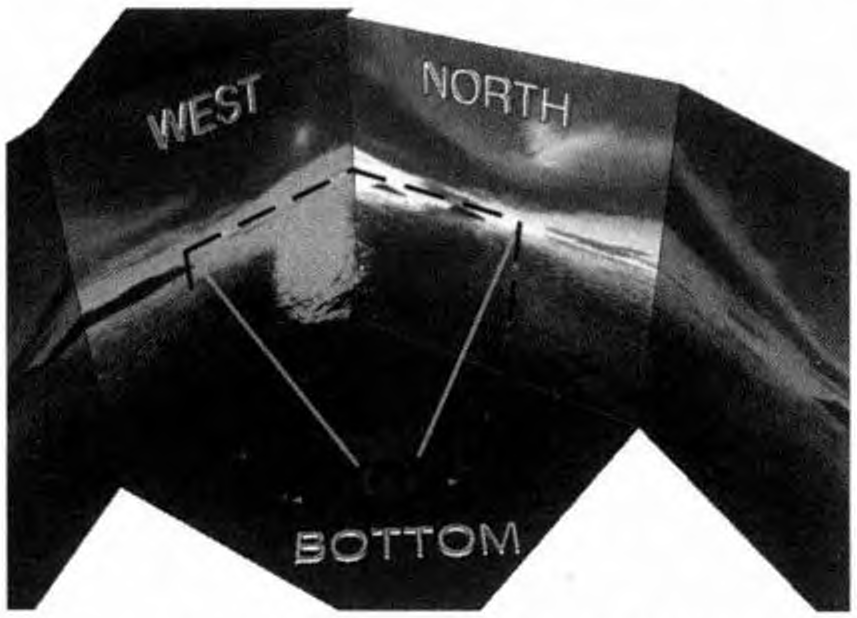


图 8-2　立方体纹理的纹素颜色取样

当假想摄像头改变方向时，来自摄像头的矢量也会与立方体的不同面相交。当摄像头竖直向上时，可能只会从立方体的顶面纹理中取样。当摄像头从图 8-2 中的方向开始顺时针旋转 90 度后，可能就要从北面、东面，和底面纹理取样了。

在 iOS 5 应用中，苹果的 GLKit 为天空盒的渲染提供了一个专门的类 GLKSkyboxEffect。例子 OpenGLES_Ch8_1 就是使用 GLKSkyboxEffect 类产生了图 8-1 中的屏幕截图。这个例子会让视点在天空盒中绕着帆船模型转动，以此来展示效果。

下面摘录自 OpenGLES_Ch8_1ViewController 类的代码就是分配、初始化，并配置一个天空盒的步骤。

```
// Create and configure skybox
self.skyboxEffect = [[GLKSkyboxEffect alloc] init];
self.skyboxEffect.textureCubeMap.name = self.textureInfo.name;
self.skyboxEffect.textureCubeMap.target =
   self.textureInfo.target;
self.skyboxEffect.xSize = 6.0f;
self.skyboxEffect.ySize = 6.0f;
self.skyboxEffect.zSize = 6.0f;
```

天空盒的尺寸是随意的。它需要足够大以容纳要渲染的场景，并且它需要是一个立方体（所有的面尺寸相等），以避免渲染时的纹理扭曲。天空盒的中心要在绘图前设定，并且必须要尽量靠近用于设置视点的眼睛位置。如果眼睛的位置太接近天空盒立方体的边缘，那么由此产生的纹理拉伸会损坏最终的渲染效果。

GLKTextureLoader 提供了处理 GLKSkyboxEffect 所需的立方体纹理贴图的方法。下面的粗体代码使用一个名为 skybox0.png 的图像作为立方体纹理贴图，并初始化了 self.textureInfo。

```
// Load cubeMap texture
NSString *path = [[NSBundle bundleForClass:[self class]]
   pathForResource:@"skybox0" ofType:@"png"];
NSAssert(nil != path, @"Path to skybox image not found");
NSError *error = nil;
self.textureInfo = [GLKTextureLoader
   cubeMapWithContentsOfFile:path
   options:nil
   error:&error];
```

为了绘制天空盒，需要设置 GLKSkyboxEffect 实例的中心等于眼睛的位置，并设置 projectionMatrix 和 modelViewMatrix，以匹配用于渲染场景其他部分的变换。调用天空盒的“-prepareToDraw”方法，然后调用天空盒的“-draw”方法。例子 OpenGLES_Ch8_1ViewController 就是使用下面的代码做到这些的。

```
// Draw skybox centered on eye position
self.skyboxEffect.center = self.eyePosition;
self.skyboxEffect.transform.projectionMatrix =
   self.baseEffect.transform.projectionMatrix;
self.skyboxEffect.transform.modelviewMatrix =
   self.baseEffect.transform.modelviewMatrix;
[self.skyboxEffect prepareToDraw];
[self.skyboxEffect draw];
```

注意 与天空盒类似，还有一个效果叫做环境贴图（environment map），又叫做反射贴图（reflection map），使用它可以很容易地模拟镜面和类似水面的其他反射面。GLKit 包含 GLKReflectionMapEffect 类。你的应用会像 GLKBaseEffect 实例一样配置关于镜面反射的属性，并且会多设置一个属性——textureCubeMap。渲染反射与渲染天空盒的操作相同。两种情况下都是使用方向来取样纹理的。对于天空盒，这个方向是从观察者到包裹立方体的面。对于环境贴图，这个方向是指从反射面到包裹立方体的面。

8.2 深入探讨 GLKSkyboxEffect 是怎么工作的

例子 OpenGLES_Ch8_2 中的 AGLKSkyboxEffect 实现了 GLKit 的 GLKSkyboxEffect，并展示了怎么使用 OpenGL ES 2.0 建立一个天空盒效果。虽然例子 OpenGLES_Ch8_2 使用的是 AGLKSkyboxEffect 而不是 GLKSkyboxEffect，但是除此之外它与例子 OpenGLES_Ch8_1 是相同的。

下面是 AGLKSkyboxEffect 的接口代码，与 GLKSkyboxEffect 相似。

AGLKSkyboxEffect has the following interface similar to **GLKSkyboxEffect**:

```
//
//  AGLKSkyboxEffect.h
//
//

#import <GLKit/GLKit.h>

@interface AGLKSkyboxEffect : NSObject <GLKNamedEffect>

@property (nonatomic, assign) GLKVector3 center;
@property (nonatomic, assign) GLfloat xSize;
@property (nonatomic, assign) GLfloat ySize;
@property (nonatomic, assign) GLfloat zSize;
@property (strong, nonatomic, readonly) GLKEffectPropertyTexture
   *textureCubeMap;
@property (strong, nonatomic, readonly) GLKEffectPropertyTransform
   *transform;

- (void) prepareToDraw;
- (void) draw;

@end
```

AGLKSkyboxEffect 会使用下面的顶点所定义的三角形带绘制天空盒立方体。立方体每个面的长度是 1.0（–0.5 到 0.5）。使用这个尺寸可以方便地把天空盒缩放为任意的大小。

```
// The 8 corners of a cube
const float vertices[AGLKSkyboxNumCoords] = {
    -0.5, -0.5,  0.5,
     0.5, -0.5,  0.5,
    -0.5,  0.5,  0.5,
     0.5,  0.5,  0.5,
    -0.5, -0.5, -0.5,
     0.5, -0.5, -0.5,
    -0.5,  0.5, -0.5,
     0.5,  0.5, -0.5,
};
```

```
glGenBuffers(1, &vertexBufferID);
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferID);
glBufferData(GL_ARRAY_BUFFER,
   sizeof(vertices),
   vertices,
   GL_STATIC_DRAW);

// Indices of triangle strip to draw cube
// Order is critical to make "front" faces be on inside
// of cube.
const GLubyte indices[AGLKSkyboxNumVertexIndices] = {
   1, 2, 3, 7, 1, 5, 4, 7, 6, 2, 4, 0, 1, 2
};
glGenBuffers(1, &indexBufferID);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER,
   sizeof(indices),
   indices,
   GL_STATIC_DRAW);
```

AGLKSkyboxEffect 会使用与前面的例子相似的代码来利用一个顶点属性数组缓存和一个元素数组缓存绘制立方体。绘制天空盒的诀窍是确保三角形带中的每个三角形的前面朝向天空盒内部。否则，从天空盒内部渲染场景时就会产生错误的结果。如在第7.3节所讲解的，每个三角形的顶点顺序决定了哪一面是前面。

绑定天空盒纹理为绘制做的准备。然后平移天空盒的坐标系，直到天空盒的中心处在 center 属性所指定的位置，最后使用 xSize、ySize 和 zSize 属性所指定的尺寸缩放天空盒的坐标系。这些变换会施加到天空盒的 modelviewMatrix 上，在天空盒准备好绘制时 modelviewMatrix 矩阵已经包含了视点的变换。

```
GLKMatrix4 skyboxModelView = GLKMatrix4Translate(
   self.transform.modelviewMatrix,
   self.center.x, self.center.y, self.center.z);
skyboxModelView = GLKMatrix4Scale(
   skyboxModelView,
   self.xSize, self.ySize, self.zSize);
```

接着绑定并配置天空盒的顶点属性数组缓存和元素数组缓存。

```
glEnableVertexAttribArray(GLKVertexAttribPosition);
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferID);
glVertexAttribPointer(GLKVertexAttribPosition,
   3,
   GL_FLOAT,
   GL_FALSE,
   0,
   NULL);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferID);
```

AGLKSkyboxEffect 的“-draw”方法会利用 glDrawElements 函数向 GPU 发送顶点数据，具体代码如下：

```
- (void)draw
{
   glDrawElements(GL_TRIANGLE_STRIP,
      AGLKSkyboxNumVertexIndices,
      GL_UNSIGNED_BYTE,
      NULL);
}
```

天空盒的 Shading Language 程序

GLKit 会自动生成 Shading Language 程序，因此通常不需要你自己编写。尽管如此，在前面的例子中已经使用了多个自定义的 OpenGL ES Shading Language 程序。第 3 章的例子 OpenGLES_Ch3_6 改编了苹果的一个示例程序来渲染和驱动两个相似的立方体：其中一个立方体是使用 GLKit 的 GLKBaseEffect 绘制的，另一个是使用一个自定义的 Shading Language 程序绘制的，这个自定义的 Shading Language 程序复制了 GLKBaseEffect 生成的 Shading Language 程序的部分功能。

第 5 章的例子 OpenGLES_Ch5_5 创建了 AGLKTextureTransformBaseEffect 类及其相关的 Shading Language 程序，以此来演示 iOS 5 的 GLKit 还不支持的纹理坐标变换功能。第 6 章的多个例子扩展了 AGLKTextureTransformBaseEffect 类，以此来模拟聚光灯，这个聚光灯与 OpenGL ES 1.1 内置的以及 GLKit 复制的聚光灯功能相似。实际上，AGLKTextureTransformBaseEffect 类中使用的这个自定义 Shading Language 程序超过了 iOS 5 中的 Shading Language 程序的实际复杂度。第 7 章介绍了 AGLKArmatureBaseEffect 类以及与之相伴的相对简单的 Shading Language 程序。

本节提供了一个揭秘自定义 Shading Language 程序的旋风之旅。这个主题是庞大且复杂的，一些书对这个主题做了详细的讲解，比如由 Aaftab Munshi、Dan Ginsburg 和 Dave Shreiner 所编著的《OpenGL ES 2.0 Programming Guide》一书。除非确实需要使用 OpenGL ES 2.0，否则苹果的 GLKit 会减少并通常会尽量消除 Shading Language 代码，但实际上苹果无法提供你所需的所有 Shading Language 程序。在重新构建 GLKSkyboxEffect 的过程中，例子 OpenGLES_Ch8_2 的 AGLKSkyboxEffect 类会使用一个足够简单的 Shading Language 程序做一个详细介绍。

OpenGL ES 2.0 Shading Language 程序通常包含两个着色器（shader）子程序。一个是顶点着色器（vertex shader），在渲染一个场景时，每处理一个顶点这个着色器都会执行一次。另一个着色器是片元着色器（fragment shader），这个着色器会为每个可能写入像素颜色渲染缓存的片元执行一次。一般场景通常包含数十万或者数百万个顶点。每次渲染一个场景时，GPU 都可以计算一百万个或者更多的片元颜色。回想一下，当几何对象重叠时就会出现片元重复。深度缓存可以决定重叠片元中的哪些片元会决定像素颜色渲染缓存

中的最终值。否则的话，为每个位置最后计算的片元颜色会替换掉前面计算的颜色值。

现代 GPU 是非常快速并且高度并行的。即便如此，简单的 Shading Language 程序通常仍然比复杂的程序执行得快。当程序每秒执行六千万或者更多次时，简单变得非常重要。据粗略计算，一个包含 100 个指令并要计算 100 万个片元颜色的片元着色器要产生每秒 60 帧的渲染结果的话，就需要 GPU 在一秒内执行 60 亿个指令。

1. 顶点着色器

顶点着色器会为提交到 GPU 的每个顶点运行一次，并且每次顶点着色器运行时，它必须要设置一个叫做 gl_Position 的内置变量的值。赋给 gl_Position 的值指定了像素颜色渲染缓存的坐标系中的一个位置，这个坐标系有时又被称为剪裁空间（clip space），因为渲染缓存坐标系之外的值是不可见的，并且因此会被从渲染结果中“剪裁出去”。顶点着色器可以自由地使用任意逻辑来计算 gl_Position 的值，但是通常情况下，传入的顶点位置属性是通过 model-view 矩阵和投影矩阵的变换来计算像素颜色渲染缓存中的对应位置的，就像在第 5 章中讲解的那样。例子 OpenGLES_Ch8_2 使用的就是下面的顶点着色器。粗体代码显示的就是对 gl_Position 变量的赋值语句。

```
//
//  AGLKSkyboxShader.vsh
//

/////////////////////////////////////////////////////////////////
// VERTEX ATTRIBUTES
/////////////////////////////////////////////////////////////////
attribute vec3 a_position;

/////////////////////////////////////////////////////////////////
// UNIFORMS
/////////////////////////////////////////////////////////////////
uniform highp mat4      u_mvpMatrix;
uniform sampler2D       u_unitCube[1];

/////////////////////////////////////////////////////////////////
// Varyings
/////////////////////////////////////////////////////////////////
varying lowp vec3       v_texCoord[1];


void main()
{
   v_texCoord[0] = a_position;
   gl_Position = u_mvpMatrix * vec4(a_position, 1.0);
}
```

分解整个顶点着色器，首先是声明每个顶点所需的属性。这个着色器只使用一个属性，就是每个顶点的位置属性。OpenGL ES Shading Language 的关键字 attribute 用于声明一个属性，并且会指定属性的数据类型。在当前情况下，语句“attribute vec3 a_position;”声明了一个命名为 a_position 的属性，这个属性是由三个浮点元素组成的矢量。属性的名字是完全随意的。属性可以被顶点程序以任何方式使用。当前的 Shading Language 程序需要一个什么样的位置属性完全取决于为 GPU 提供顶点属性数组缓存的程序员。根本没有或者有很少编译器支持或运行时检查。一个不明智的程序员可能会为 GPU 提供顶点颜色属性而不是位置属性，这样导致 GPU 渲染垃圾，而不会做任何检测或者错误报告。OpenGL ES Shading Language 可能看起来有点像 C 语言，但它实际上是一种标准化的 GPU 汇编语言形式，正如一些程序员说的，你正在以非常接近硬件的层次编程。顶点属性只在顶点着色器中可用。片元着色器并不能访问它们。

uniform 变量会保存在 GPU 控制的内存中。在向 GPU 提供顶点属性之前必须要设置它们的值，在所有提交的顶点被处理之前这些值可能都不会发生变化。uniform 关键字会声明统一的变量，并且通常还会设置精度和数据类型。语句“uniform highp mat4 u_mvpMatrix;”声明了一个命名为 u_mvpMatrix 的统一变量，这个变量是一个高精度浮点值的 4×4 矩阵。高精度值相当于 C 语言中的 GLfloat 数据类型。除了 highp，还有两个其他的可用精度，mediump 和 lowp。OpenGL ES 2.0 标准并没有指定 mediump 和 lowp 所代表的具体精度，除了 lowp 变量必须能够保存颜色元素值，以及 mediump 变量必须要支持 lowp 和 highp 之间的一些精度。对于 iOS 设备来说，lowp 变量会保存在 –1.0 到 1.0 范围内的浮点值，并且这将足以保存标准化矢量和颜色矢量。

在大多数情况下，GPU 使用 lowp 变量进行运算的速度要比使用 highp 变量的速度更快，因此应该尽量使用最小的精度来满足每个程序的需求。类似这个例子中使用的 u_mvpMatrix 变量的变换矩阵需要保存超出 –1.0 到 1.0 范围的值，因此必须要声明为 highp 类型。

在这个例子中使用的其他统一变量会保存一个纹理取样单元的标识符，纹理取样单元又被称为纹理单元。

```
uniform samplerCube        u_unitCube[1];
```

samplerCube 数据类型是另一个内置的魔法位。每个 GPU 都有两个或者多个通常并行工作的硬件纹理取样单元。u_unitCube 变量声明了一个用来在 GPU 的纹理单元之间做选择的标识符数组。这个变量被声明为了一个统一变量，因此可以在向 GPU 提交顶点属性之前通过编程改变它的值。这个程序只需要一个纹理取样单元，因此 u_unitCube 是一个只存储了一个标识符的数组。

天空盒使用的是 8.1 节讲解的立方体纹理取样方式。更常见的 2D 位置纹理取样方式是使用内置的 sampler2D 数据类型来指定纹理取样单元的。实际上，对于 GLuint 来说，samplerCube 和 sampler2D 是相同的。两种数据类型都对应于 OpenGL ES 的资源标识符。

第三个也是最后一种变量是使用 varying 关键字来声明的。可变变量是可选的附加

输出变量，用于顶点着色器和片元着色器间的数据传递。与统一变量类似，可变变量有指定的精度。片元着色器无法访问顶点属性，因此可变变量的一个常见用途是保存从顶点着色器获得的属性。大部分 GPU 会在稀缺的 GPU 寄存器中存储可变变量。varying 标识符只能用在 float、vec2、vec3、vec4、mat2、mat3 和 mat4 数据类型中，或者这些类型的数组中。OpenGL ES 2.0 标准需要 GPU 支持至少 32 个浮点类型的可变变量。一个 vec2 记作两个浮点值。一个 mat4 记作 16 个浮点值。

例子 OpenGLES_Ch8_2 的顶点程序会使用 varying lowp vec3 v_texCoord[1] 数组向片元程序传递纹理坐标。这个变量的精度是 lowp，因为一个天空盒的纹理坐标通常在 –1.0 到 1.0 的 lowp 精度范围内。一个更加通用的着色器不应该做这个假设而应该使用 highp 精度的纹理坐标。这个数组只包含一个纹理坐标集，因为在这个例子中只使用了一个纹理。一个多重纹理片元着色器可能为每个顶点使用多个纹理坐标。

所有的 Shading Language 着色器都有一个 main() 函数，GPU 会使用这个函数作为着色器的入口点。在这个例子的顶点着色器中的 main() 函数是尽可能简单的。每个未变换的顶点位置属性还可以用作天空盒立方体内的纹理坐标。这只是本例的特定“单元立方体”顶点位置所引起的一个实现细节。纹理坐标通常是作为顶点属性来存储的，并且是与位置属性分离的。

```
void main()
{
   v_texCoord[0] = a_position;
   gl_Position = u_mvpMatrix * vec4(a_position, 1.0);
}
```

依照所有顶点着色器所需要的，设置内置变量 gl_Position。统一变量 u_mvpMatrix 保存了定义了视点的 model-view 矩阵和投影矩阵的结合。使用 u_mvpMatrix 变化 a_position 属性以得出像素颜色渲染缓存坐标系内的对应位置。表达式 vec4(a_position, 1.0) 会把三维位置矢量扩展成四维矢量。设置第四个坐标为 1.0 会让接下来的变换可以包含在 u_mvpMatrix 中编码的任意平移。如果第四个坐标被设置成 0.0，那么这个位置可以旋转和缩放，但是不能平移。

2. 片元着色器

理解片元着色器的关键点之一是要意识到片元与顶点着色器输出的位置并不是一一对应的。转换几何图形数据为帧缓存内的颜色像素的这个渲染步骤叫做栅格化（rasterizing）。为了栅格化几何图形，GPU 会跟踪要渲染的点、线段，或者三角形是由哪些顶点定义的。然后 GPU 会使用片元着色器计算由每个点、线段，或者三角形所产生的每个片元的颜色。会计算出顶点本身位置以及顶点之间的所有位置的片元颜色。

GPU 会为顶点着色器所设置的所有可变变量插值。例如，顶点着色器会为定义了一条线段的两个顶点各执行一次，并且每次顶点着色器运行时都会在 gl_Position 中

存储一个不同的像素颜色渲染缓存位置。GPU 必须要计算分别赋值给 gl_Position 的两个顶点位置之间的线段上的片元颜色。通过使用线性内插法来做到这一点，以使在两个顶点之间的一个位置近似等于每个末端顶点位置的平均值，{（x0*0.5+x1*0.5），（y0*0.5+y1*0.5）}。在第一个顶点和第二个顶点之间的 3/4 位置处的值是这样计算的：{(x0 * 0.25 + x1 * 0.75), (y0 * 0.25 + y1 * 0.75)}。实际上，这个计算并不总是那么简单，因为透视所产生的透视收缩会影响顶点间的片元的内插位置。

片元着色器会为可变变量值的每个内插集合执行一次。并不是只有位置是内插值的，纹理坐标和颜色也是内插值的。当在一个片元程序中计算每个片元的灯光时，即使是法向量也是在顶点间内插值的。

下面的代码实现了例子 OpenGLES_Ch8_2 的片元着色器。就像顶点着色器会设置特定的内置的 gl_Position 变量一样，片元着色器必须要设置内置的 gl_FragColor 变量为每个片元计算出来的最终颜色。如下代码中的粗体代码执行的就是这个赋值。

```
//
//  AGLKSkyboxShader.fsh
//
//

/////////////////////////////////////////////////////////////////
// UNIFORMS
/////////////////////////////////////////////////////////////////
uniform highp mat4      u_mvpMatrix;
uniform samplerCube     u_unitCube[1];

/////////////////////////////////////////////////////////////////
// Varyings
/////////////////////////////////////////////////////////////////
varying lowp vec3       v_texCoord[1];


void main()
{
   gl_FragColor = textureCube(u_unitCube[0], v_texCoord[0]);
}
```

片元着色器是尽可能简单的。最终片元颜色是通过调用内置的 textureCube() 函数计算出来的，这个函数首先会指定要使用的纹理取样单元的标识符，然后会指定片元的纹理坐标。请记住，这个纹理坐标就是 GPU 插补的实际值。

编写 OpenGL ES 2.0 Shading Language 程序有很多的细微差别、局限和古怪。OpenGL ES 2.0 Shading Language 是桌面版 OpenGL Shading Language 的一个子集，因此要谨慎对待适用于桌面系统的例子。在网址：http://www.khronos.org/opengles/sdk/docs/reference_cards/OpenGL-ES-2_0-Reference-card.pdf 上有一个方便的快速参考卡，如果你是一个新手的话，这个快速参考卡可以帮助你找到内置函数、类型、精度和内置

变量的正确名字。

3. 在你的代码中使用 OpenGL ES 2.0 Shading Language 程序

苹果提供的实例代码会包含下面的四个方法，本书中的每个自定义 Shading Language 例子都重用了这四个方法。

```
- (BOOL)loadShaders;
- (BOOL)compileShader:(GLuint *)shader
   type:(GLenum)type
   file:(NSString *)file;
- (BOOL)linkProgram:(GLuint)prog;
- (BOOL)validateProgram:(GLuint)prog;
```

这些方法指定了很多在图 8-3 中显示的用来创建、编译、连接和检验 Shading Language 程序的流程步骤。

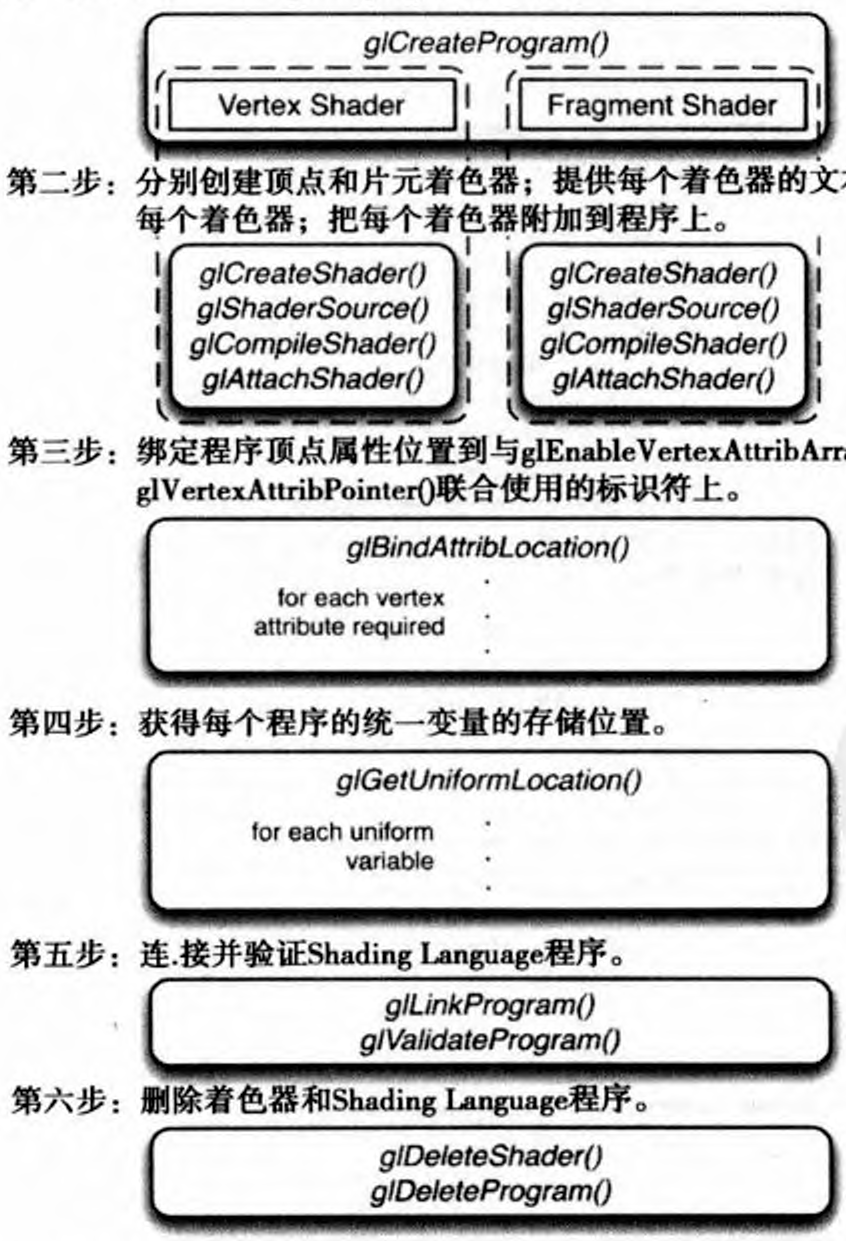


图 8-3 创建 OpenGL ES 2.0 Shading Language 程序的步骤

完整地介绍创建 OpenGL ES 2.0 Shading Language 程序对象的步骤会超出本书的范围。但是其实并没有看起来那么难。借助苹果的例子方法，只要包含下面的代码到类似 AGLKSkyboxEffect、AGLKTextureTransformBaseEffect，或者 AGLKArmatureBaseEffect 的自定义类的“-prepareToDraw”方法中就可以了，就是这么简单。

```
if(0 == program)
{
   [self loadShaders];
}
```

本书每个例子中的“-loadShaders”方法都是相同的，除了包含顶点和片元着色器源代码的文件的名字、特定属性，以及着色器使用的统一变量。检查例子 OpenGLES_Ch8_2 中的 AGLKSkyboxEffect 类的“-loadShaders”方法，这是“-loadShaders”方法的一个具有代表性的实现。

在程序编译、连接，并验证之后，图 8-4 列出了使用一个自定义 Shading Language 程序的步骤。

图 8-4　使用一个编译的 OpenGL ES 2.0 Shading Language 程序的步骤

AGLKSkyboxEffect、AGLKTextureTransformBaseEffect 和 AGLKArmatureBaseEffect 以及 GLKit 的 GLKBaseEffect、GLKSkyboxEffect 和 GLKReflectionMapEffect 类的“-prepareToDraw”和“-draw”方法联合执行了图 8-4 中列出的所有步骤。下面代码显示了在简单的 AGLKSkyboxEffect 类中的“-prepareToDraw”和“-draw”方法的完整实现。粗体代码突出显示了每个步骤。

```
- (void) prepareToDraw
{
   if(0 == program)
   {
      [self loadShaders];
   }

   if(0 != program)
   {
      glUseProgram(program);                    // Step 1

      // Translate skybox cube to specified center and scale to
      // specified size
      GLKMatrix4 skyboxModelView = GLKMatrix4Translate(
         self.transform.modelviewMatrix,
         self.center.x, self.center.y, self.center.z);
      skyboxModelView = GLKMatrix4Scale(
         skyboxModelView,
         self.xSize, self.ySize, self.zSize);

      // Pre-calculate the mvpMatrix
      GLKMatrix4 modelViewProjectionMatrix = GLKMatrix4Multiply(
         self.transform.projectionMatrix,
         skyboxModelView);

      // Standard matrices
      glUniformMatrix4fv(uniforms[AGLKMVPMatrix], 1, 0,
         modelViewProjectionMatrix.m);          // Step 2

      // One texture sampler
      glUniform1i(uniforms[AGLKSamplersCube], 0);// Step 2

      glEnableVertexAttribArray(GLKVertexAttribPosition);
      glBindBuffer(GL_ARRAY_BUFFER, vertexBufferID);
      glVertexAttribPointer(GLKVertexAttribPosition,
         3,
         GL_FLOAT,
         GL_FALSE,
         0,
         NULL);                                 // Step 3
```

```
        glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, indexBufferID);

        // Bind the texture to be used
        if(self.textureCubeMap.enabled)
        {
           glBindTexture(GL_TEXTURE_CUBE_MAP,
              self.textureCubeMap.name);              // Step 4
        }
        else
        {
           glBindTexture(GL_TEXTURE_CUBE_MAP, 0);  // Step 4
        }
     }
}


- (void)draw
{
   glDrawElements(GL_TRIANGLE_STRIP,
      AGLKSkyboxNumVertexIndices,
      GL_UNSIGNED_BYTE,
      NULL);                                          // Step 5
}
```

如果本节对于 AGLKSkyboxEffect 类的实现的深入探讨让你迷惑，请不要担心。GLKit 以及类似 GLKBaseEffect 和 GLKSkyboxEffect 类的存在就是为了让你不必了解 OpenGL ES 2.0 Shading Language 的错综复杂，也不必了解使用 Shading Language 程序的所有相关步骤。重要的是要学习吸收图形的概念，当需要的时候再回来了解实现细节。自定义着色语言程序与在其他编程环境中的自定义汇编语言几乎完全一致。并不是每个人都需要了解如此底层的实现，但是有些控制只有在最底层才可以实行。

下一节包含一个用于管理另一个特效（粒子）的自定义 Shading Language 程序。试着找出粒子程序与 AGLKSkyboxEffect Shading Language 程序的相似处。

8.3 粒子

在计算机图形中，粒子通常指的是任意的有颜色的或者有纹理的几何对象，这个几何对象可以用在 3D 视域中的一个单独的位置来定义。应用通常要每次渲染数百或者数千个粒子，这让高效渲染变得非常重要。应用会快速重设粒子的位置以产生逼真的动画。图 8-5 显示的是例子 OpenGLES_Ch8_3 所演示的多个

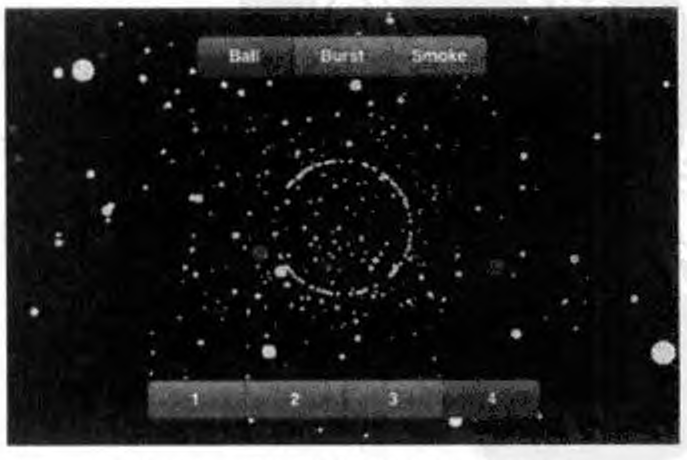


图 8-5　粒子效果的屏幕截图

粒子效果之一。测试一下这个例子，感受一下动画效果。

与所有其他 GPU 渲染的图形一样，可以使用点、线段，或者三角形来实现粒子效果。使用由很多三角形组成的几何球体来实现屏幕上的每个粒子，这样的实现粒子效果的方式要付出非常高昂的计算代价。从概念上讲，球体可以做出理想的粒子效果，因为它们是使用一个单独的点来定位的，即球体的中心位置，同时它们还具有径向对称性，这意味着不管从哪一个视点观看，它们的图形都是相同的。

幸运的是，这个球面粒子效果可以使用与视平截体的近面和远面平行的纹理矩形来创建。每次渲染场景时，都是使用视平截体来定义 3D 可见视域的，如 5.6 节所述。与近面和远面平行的矩形是将要在 8.4 节讲解的公告牌类型。不过，OpenGL ES 还包含一个叫做点精灵（point sprites）的功能，这个甚至要比用两个三角形定义的公告牌矩形渲染起来更高效。

当你用 glDrawArrays() 或者 glDrawElements() 函数指定 GL_POINTS 模式后，OpenGL ES 2.0 就会渲染点精灵。点精灵会产生以这个点精灵的位置为中心的正方形内的每个像素颜色渲染缓存位置的片元。点精灵正方形的长和宽等于在像素颜色渲染缓存坐标系中的当前点精灵的尺寸。不幸的是，自定义 Shading Language 程序需要控制每个点精灵的尺寸。

例子 OpenGLES_Ch8_3 创建了 AGLKPointParticleEffect 类，这个类封装了所需的自定义 Shading Language 程序，并提供了控制粒子效果的其他选项。每个粒子都有一个初始位置、速度和尺寸。每个粒子的力向量都会随着时间改变粒子的速度。模拟的全局重力也会影响每个粒子的速度。最后，每个粒子都会随着时间褪色，直到完全透明，并且每个粒子都有一个寿命，超过这个寿命后，这个粒子就不再绘制了。

下面的代码声明了 AGLKPointParticleEffect 类的接口。

The following code declares the interface for **AGLKPointParticleEffect**:

```
//
//  AGLKPointParticleEffect.h
//  OpenGLES_Ch8_3
//

#import <GLKit/GLKit.h>

////////////////////////////////////////////////////////////////////////
// Default gravity acceleration vector matches Earth's
// {0, (-9.80665 m/s/s), 0} assuming +Y up coordinate system
extern const GLKVector3 AGLKDefaultGravity;

@interface AGLKPointParticleEffect : NSObject  <GLKNamedEffect>

@property (nonatomic, assign) GLKVector3 gravity;
@property (nonatomic, assign) GLfloat elapsedSeconds;
```

```
@property (strong, nonatomic, readonly) GLKEffectPropertyTexture
   *texture2d0;
@property (strong, nonatomic, readonly) GLKEffectPropertyTransform
   *transform;

- (void)addParticleAtPosition:(GLKVector3)aPosition
   velocity:(GLKVector3)aVelocity
   force:(GLKVector3)aForce
   size:(float)aSize
   lifeSpanSeconds:(NSTimeInterval)aSpan
   fadeDurationSeconds:(NSTimeInterval)aDuration;

- (void)prepareToDraw;
- (void)draw;

@end
```

AGLKPointParticleEffect 的工作方式与 GLKBaseEffect 和到目前为止出现的其他“effects”类似。texture2d0 属性定义了用于每个粒子的纹理。transform 属性存储了由 transform.projectionMatrix 和 transform.modelviewMatrix 所定义的视点。两个新属性 gravity 和 elapsedSeconds 用于控制支配粒子运动的物理引擎。gravity 属性默认与地球的重力加速度相匹配：-9.80665（下落物体的速度每秒增加 9.8 米 / 秒）。在准备一个用于绘图的 AGLKPointParticleEffect 实例之前，必须要设置 elapsedSeconds 属性。粒子的当前位置是基于连续增加的 elapsedSeconds 属性计算出来的。

与其他“effects”一样，你必须要设置属性，调用“-prepareToDraw”方法，然后调用“-draw”方法。下面摘录自 OpenGLES_Ch8_3ViewController 的“-update”方法的代码会在场景重绘前更新 elapsedSeconds 属性。

```
- (void)update
{
   NSTimeInterval timeElapsed = self.timeSinceLastResume;

   self.particleEffect.elapsedSeconds = timeElapsed;
```

接着，OpenGLES_Ch8_3ViewController 的“-glkView:drawInRect:”方法会配置粒子效果的视点来匹配基本效果，并绘制粒子。

```
// Draw particles
self.particleEffect.transform.projectionMatrix =
   self.baseEffect.transform.projectionMatrix;
self.particleEffect.transform.modelviewMatrix =
   self.baseEffect.transform.modelviewMatrix;
[self.particleEffect prepareToDraw];
[self.particleEffect draw];
```

测试各种选项是感受粒子之间的相互作用力和速度的最好方式。OpenGLES_

Ch8_3ViewController已经实现了4个用于学习粒子配置选项的方法："-spawnFireRing"、"-spawnPulse"、"-spawnSparkle" 和 "–spawnBallCannon"。例如，"-spawnBallCanon" 方法会使用下面的代码来创建粒子效果，这个效果中的粒子会在X轴方向上以任意的速度发射并进入场景。一旦启动这些粒子就会以普通的重力下落，并会在3.2秒寿命的最后半秒淡去。

```
// Turn on gravity
self.particleEffect.gravity = AGLKDefaultGravity;

float randomXVelocity = -0.5f + 1.0f *
   (float)random() / (float)RAND_MAX;

[self.particleEffect
   addParticleAtPosition:GLKVector3Make(0.0f, 0.0f, 0.9f)
   velocity:GLKVector3Make(randomXVelocity, 1.0f, -1.0f)
   force:GLKVector3Make(0.0f, 9.0f, 0.0f)
   size:4.0f
   lifeSpanSeconds:3.2f
   fadeDurationSeconds:0.5f];
```

通过修改已经实现的例子方法来测试你自己的粒子动力系统。

粒子的 Shading Language 程序

AGLKPointParticleEffect会使用自定义顶点和片元着色器来部分地展示Shading Language程序的能力范围。这里的物理计算与计算机图形的基础计算一样，使用的都是线性代数矢量和矩阵计算。例子OpenGLES_Ch8_3的所有粒子物理模拟都是在GPU中计算的。

注意 除了物理模拟外，有一个叫做"通用GPU"的趋势，这个趋势是利用GPU来执行类似音频处理、雷达信号处理、蛋白质分子分析等非图形应用领域的算法。GPU执行线性代数运算的速度通常要比通用CPU的速度快。

AGLKPointParticleEffect为每个顶点定义了下面的有点非常规的属性：

```
//////////////////////////////////////////////////////////////////
// Type used to define particle attributes
typedef struct
{
   GLKVector3 emissionPosition;
   GLKVector3 emissionVelocity;
   GLKVector3 emissionForce;
   GLKVector2 size;
   GLKVector2 emissionTimeAndLife;
}
AGLKParticleAttributes;
```

顶点着色器是使用每个每顶点属性和每个统一变量值来执行物理运算的，这里的统一变量值是指用于重力、逝去时间、纹理取样标识符、model-view 矩阵与投影矩阵的结合的统一变量值。使用完全基于初始位置（粒子创建后就不再改变）、初始速度、力和逝去时间的经典牛顿物理方程式来重新计算每个粒子的位置并存储在内置的 gl_Position 变量中。每个粒子的尺寸都存储在另一个内置变量 gl_PointSize 中。如下粗体代码显示的就是所需的赋值代码：

```
//
//  AGLKParticleShader.vsh
//
//

/////////////////////////////////////////////////////////////////
// VERTEX ATTRIBUTES
/////////////////////////////////////////////////////////////////
attribute vec3 a_emissionPosition;
attribute vec3 a_emissionVelocity;
attribute vec3 a_emissionForce;
attribute vec2 a_size;
attribute vec2 a_emissionAndDeathTimes;

/////////////////////////////////////////////////////////////////
// UNIFORMS
/////////////////////////////////////////////////////////////////
uniform highp mat4      u_mvpMatrix;
uniform sampler2D       u_samplers2D[1];
uniform highp vec3      u_gravity;
uniform highp float     u_elapsedSeconds;

/////////////////////////////////////////////////////////////////
// Varyings
/////////////////////////////////////////////////////////////////
varying lowp float      v_particleOpacity;


void main()
{
   highp float elapsedTime = u_elapsedSeconds -
      a_emissionAndDeathTimes.x;

   // Mass is assumed to be 1.0, so acceleration = force (a = f/m)
   // v = v0 + at : v is current velocity; v0 is initial velocity;
   //               a is acceleration; t is elapsed time
   highp vec3 velocity = a_emissionVelocity +
      ((a_emissionForce + u_gravity) * elapsedTime);
```

```
    // s = s0 + 0.5 * (v0 + v) * t : s is current position;
    //                               s0 is initial position;
    //                               v0 is initial velocity;
    //                               v is current velocity;
    //                               t is elapsed time
    highp vec3 untransformedPosition = a_emissionPosition +
       0.5 * (a_emissionVelocity + velocity) * elapsedTime;

    gl_Position = u_mvpMatrix * vec4(untransformedPosition, 1.0);
    gl_PointSize = a_size.x / gl_Position.w;

    // if emission life > elapsed time then non-zero with maximum
    // opacity of 1.0; otherwise 0.0. Fades over a_size.y seconds
    v_particleOpacity = max(0.0, min(1.0,
       (a_emissionAndDeathTimes.y - u_elapsedSeconds) /
       max(a_size.y, 0.00001)));
}
```

每顶点的点大小会被除以像素颜色渲染缓存坐标中的粒子位置的 w 部分。w 大体相当于粒子与平截体近面之间的距离。除以 w 模拟了粒子退向远处时的基于透视的收缩。

顶点着色器还计算了每个粒子的半透明度并使用可变变量 v_particleOpacity 来传递这个值到片元着色器中。因此，在片元着色器中并没有很多工作要做。

```
//
//  AGLKParticleShader.fsh
//
//

////////////////////////////////////////////////////////////////////////
// UNIFORMS
////////////////////////////////////////////////////////////////////////
uniform highp mat4      u_mvpMatrix;
uniform sampler2D       u_samplers2D[1];
uniform highp vec3      u_gravity;
uniform highp float     u_elapsedSeconds;

////////////////////////////////////////////////////////////////////////
// Varyings
////////////////////////////////////////////////////////////////////////
varying lowp float      v_particleOpacity;


void main()
{
   lowp vec4 textureColor = texture2D(u_samplers2D[0],
      gl_PointCoord);
   textureColor.a = textureColor.a * v_particleOpacity;

   gl_FragColor = textureColor;
}
```

片元颜色是使用内置的 texture2D() 函数求出的，同时使用了由统一变量 u_samplers2D 指定的取样器和由另一个内置的 Shading Language 魔法位所提供的纹理坐标。内置只读的 gl_PointCoord 变量是一个两元素矢量，每个元素都在 0.0 到 1.0 的范围之内，并且对应于当前正在被渲染的点精灵内的片元的 {s，t} 纹理位置。

把用于确定 gl_FragColor 的 textureColor 的半透明度值乘以 v_particleOpacity 来实现粒子的半透明。

AGLKPointParticleEffect 的局限

所有使用 AGLKPointParticleEffect 实例绘制的粒子都有相同的纹理。为了渲染包含不同纹理的粒子的场景，必须要使用多个 AGLKPointParticleEffect 实例。

当使用 GPU 计算顶点位置时会表现出一个明显的缺陷。当你测试例子 OpenGLES_Ch8_3 时，你可能会注意到一些类似某些粒子上的黑色轮廓的可视瑕疵。这些瑕疵是由粒子渲染的顺序导致的。每个粒子都会与像素颜色渲染缓存中的现存内容相融合。第一个渲染的粒子会与背景颜色相融合，但是接下来的粒子会与背景颜色以及前面渲染的粒子相融合。避免瑕疵的唯一方式是根据粒子与平截体近面的距离排序粒子，并按照从远到近的顺序渲染粒子。

在 OpenGL ES 2.0 Shading Language 程序中是无法排序顶点的。顶点通常是按照它们提交给 GPU 的顺序处理的。因此，可能需要在 AGLKPointParticleEffect 类中用 CPU 计算粒子的位置，从后向前排序粒子，并向 GPU 提交正确排序的当前粒子位置，不过这个还是留给读者作为练习吧。在下一节将会讲解的公告牌就是从后向前排序的，可以借鉴公告牌来修改 AGLKPointParticleEffect 类。甚至可以使用公告牌代替点精灵来实现粒子效果，但是点精灵的效率更高。第 12 章中的 OpenGLES_Ch12_1 例子演示了一个基于公告牌的粒子效果实现方式。

8.4 公告牌

公告牌是一直面向观察者的纹理矩形。通常有两个变种：第一种，圆柱形公告牌，会面向观察者但是还会与“上”矢量保持平行。第二种，球形公告牌，就像点精灵一样总是会与平截体的近面平行。图 8-6 显示的是使用圆柱形公告牌渲染的树。

图 8-6 展示了高细节树木的假象，只有右下角的树显得像纸一样薄。当观察者沿着“上”矢量俯视公告牌时，图 8-6 中的圆柱形公告牌就会露出破绽。

图 8-7 显示的是球形公告牌。其中的很多公告牌都呈现出逼真的假象，但是近景中的树看起来就像是躺在地面上而不是站在地面上。位于图 8-7 中心的树的树干甚至完全消失了。当公告牌与近面平行时，一些树的树干会最终渲染到地面以下。

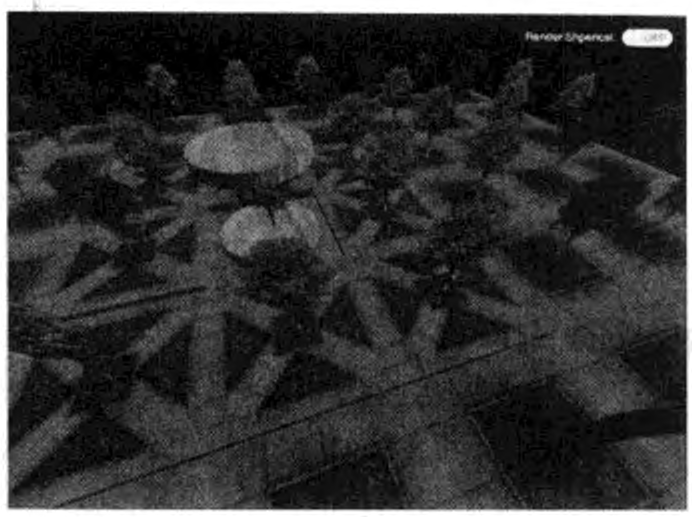


图 8-6 使用圆柱形公告牌渲染的树

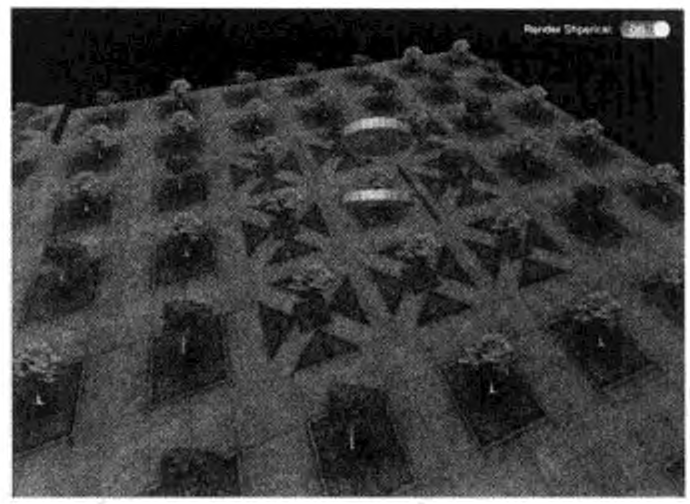


图 8-7 使用球形公告牌渲染的树木

例子 OpenGLES_Ch8_4 通过在树木之上来回飞动视点来演示公告牌效果。通过切换用户界面开关，可以在变化视点的过程中对比球形和圆柱形公告牌。两种公告牌都不能通用于所有情况。

公告牌的几个属性让它的效率远远低于点精灵。每个公告牌至少需要四个顶点，并且每次视点变化后，这些顶点的位置必须要被重新计算并重新发送给 GPU。幸运的是，计算公告牌顶点位置并不是太难，或者说并不会耗费太多的计算量。

图 8-8 阐明了定义视点的视线向量（look-at vector）与上向量之间的关系。上述信息用于使公告牌朝向观察者所需的顶点位置计算。回想一下第 4 章，两个向量的向量积定义了垂直于这两个向量的第三个向量。上向量与视线向量之间的向量积叫做“左向量”，参见图 8-8。对你来说，图 8-8 中的这个左向量可能看起来是指向右侧的，但是从看向视线向量的眼睛的视点来看，它在左边。请设想一下观察者看向纸张外部时的情景。

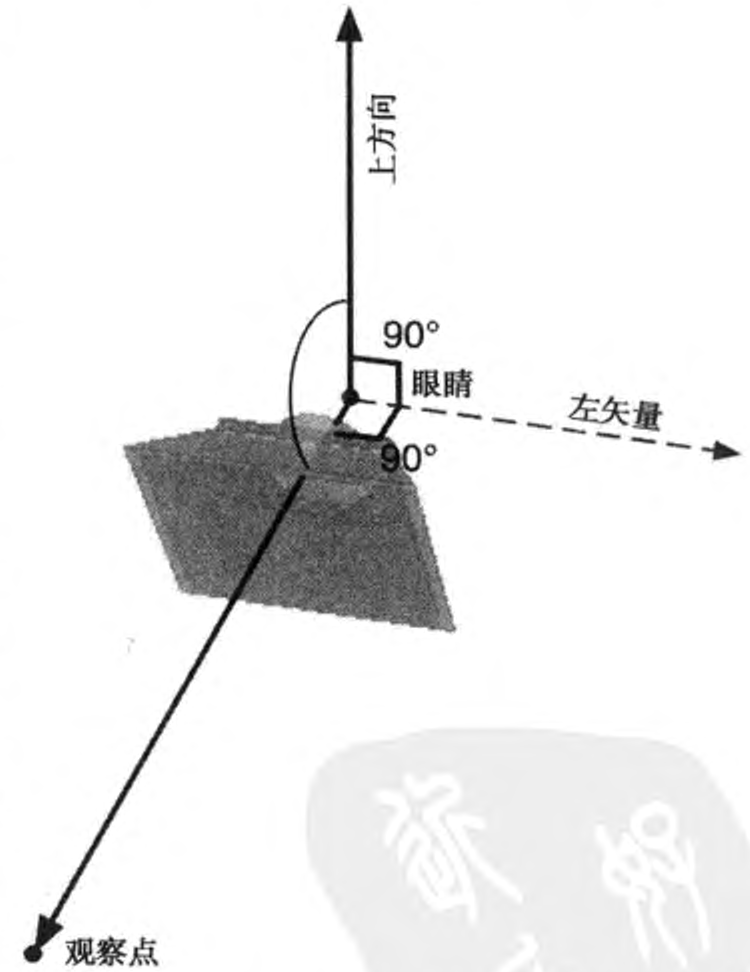


图 8-8 计算视点处的左向量

左向量定义了一个平行于视平截体的近面和远面的水平方向。但是，上向量与视线向量之间的角度没有必要是 90 度，并且没有定义近面和远面的垂直线。图 8-9 显示的是整个视平截体。“Frustum Up Vector”是视线向量和左向量的向量积。

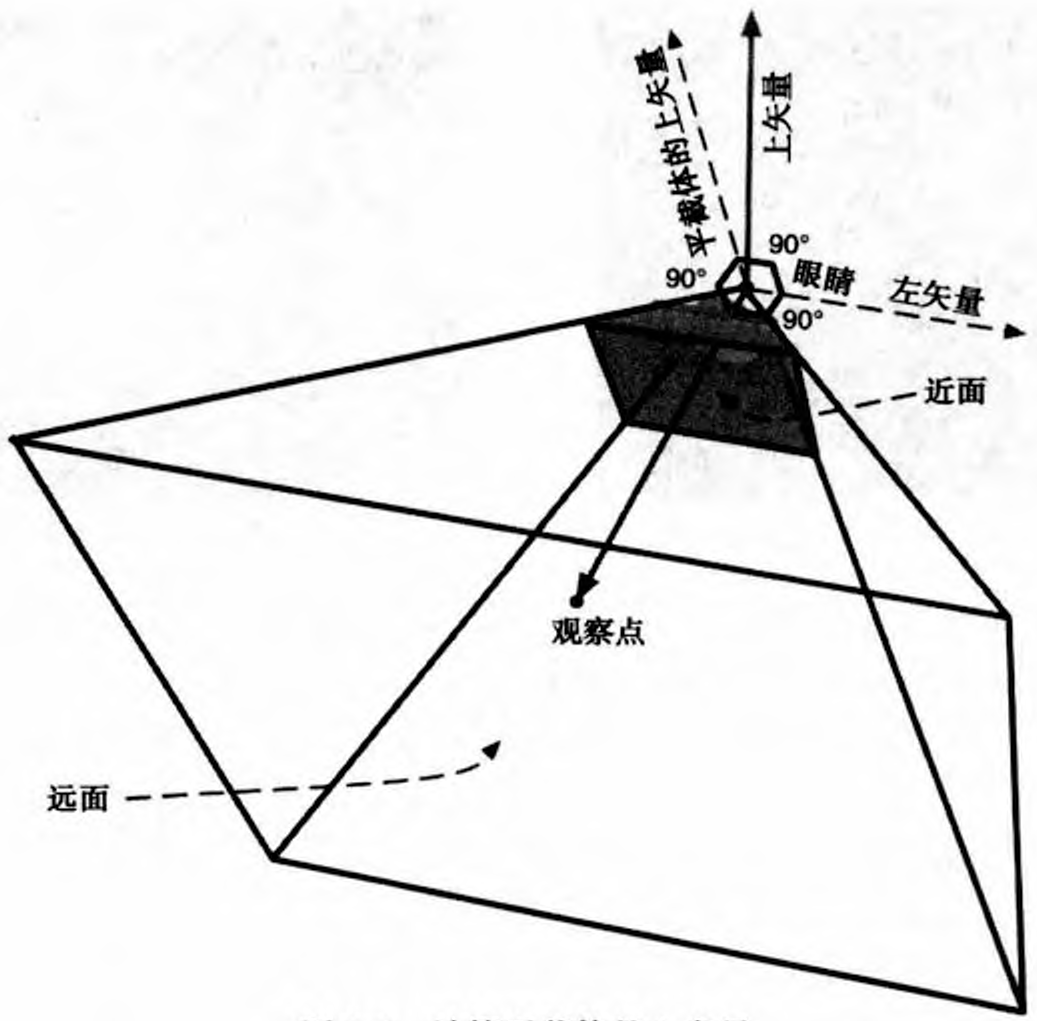


图 8-9　计算平截体的上向量

提供一个位置、一个大小和左向量，我们就可以为圆柱形公告牌计算顶点位置。图 8-10 显示了原点在底部中心的圆柱形公告牌的几何形状，这个图可以帮助可视化用于计算定义公告牌边界的顶点的方程式。

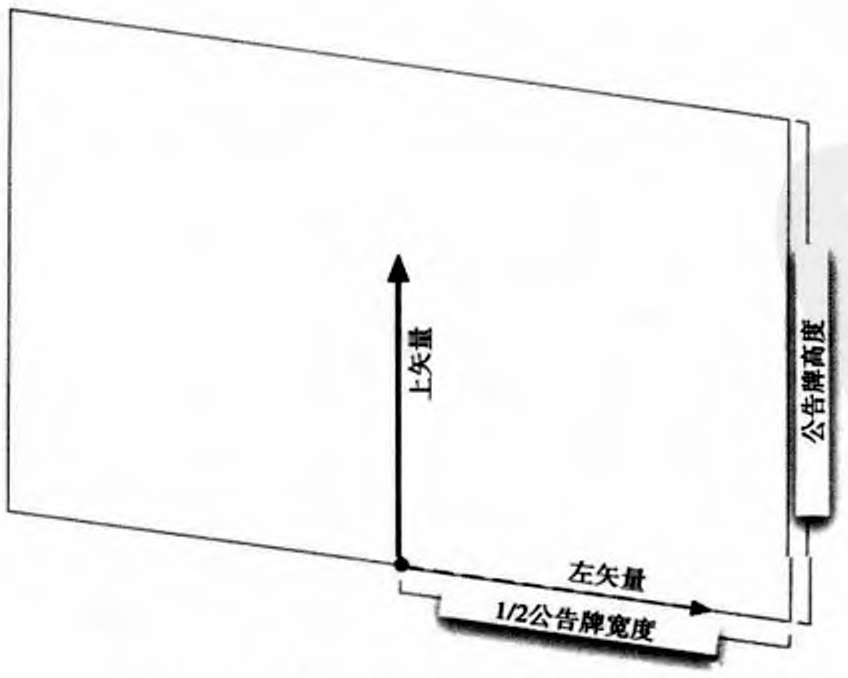


图 8-10　原点在底部中心的圆柱形公告牌

标准左向量是标准上向量和标准视线向量的向量积。

```
GLKVector3 leftVector = GLKVector3CrossProduct(
   GLKVector3Normalize(upVector),
   GLKVector3Normalize(lookAtVector));
```

左向量是标准化的，这意味着它的长度是 1.0。公告牌宽度的一半乘以左向量会产生一个长度是所需的长度同时方向与左向量相同的新矢量。左下顶点位置是通过添加这个新矢量到公告牌的底部中心来建立的。

```
GLKVector3 leftBottomPosition =
   GLKVector3Add(GLKVector3MultiplyScalar(
   leftVector, billboard.size.x * 0.5f),
   billboard.position);
```

右下顶点位置的距离是相同的，但是方向相反。

```
GLKVector3 rightBottomPosition =
   GLKVector3Add(GLKVector3MultiplyScalar(
   leftVector, billboard.size.x * -0.5f),
   billboard.position);
```

从底部顶点位置处沿着上向量的方向向上平移公告牌高度的距离就是顶部顶点的位置。

```
GLKVector3 leftTopPosition =
   GLKVector3Add(leftBottomPosition,
      GLKVector3MultiplyScalar(
         upVector, billboard.size.y));
GLKVector3 rightTopPosition =
   GLKVector3Add(rightBottomPosition,
      GLKVector3MultiplyScalar(
         upVector, billboard.size.y));
```

球形公告牌的顶点是使用平截体上向量而不是场景上向量计算出来的，参见图 8-11。例子 OpenGLES_Ch8_4 会计算平截体上向量，并临时重赋值上向量来保持所有其他的用于寻找顶点位置的方程式都相同。

```
if(self.shouldRenderSpherical)
{  // Recalculate up vector
   upVector = GLKVector3CrossProduct(
      normalizedLookAtVector,
      leftVector);
}
```

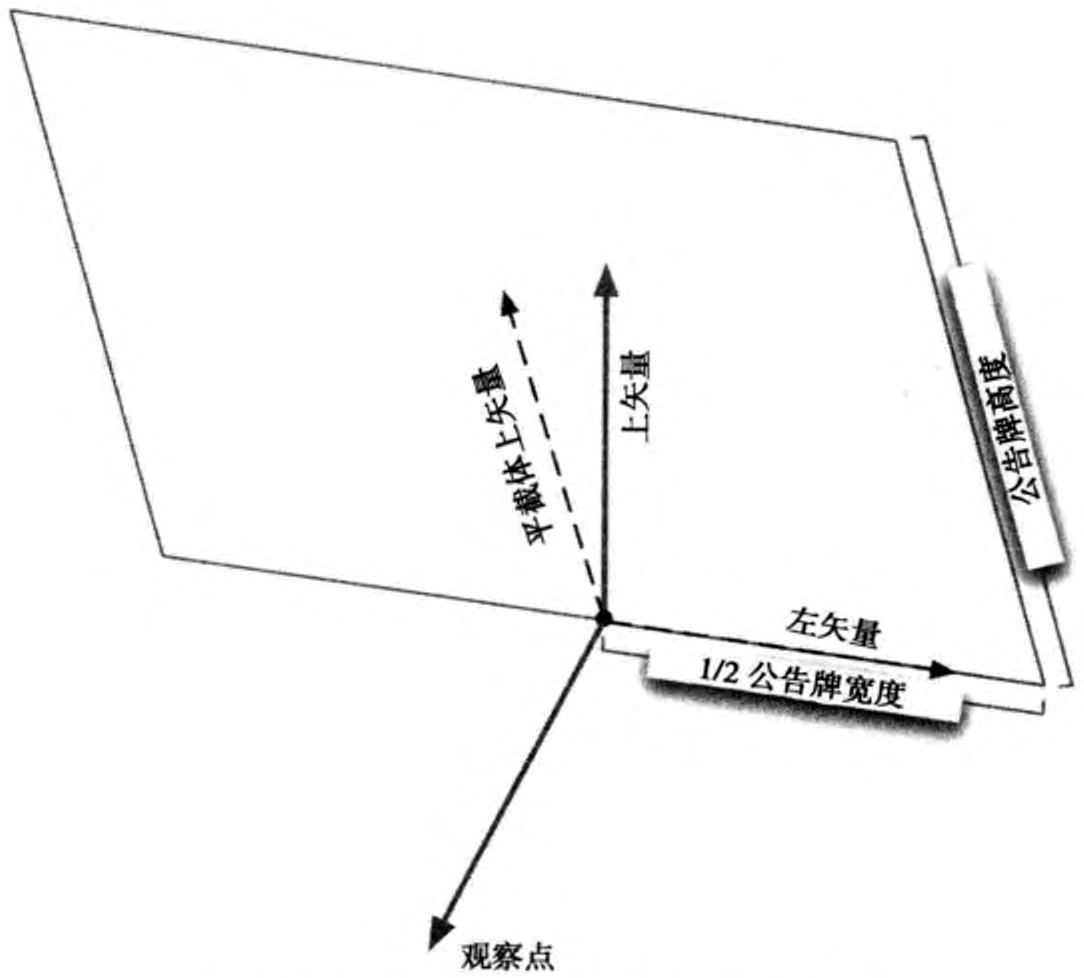


图 8-11　原点在底部中心的球形公告牌

例子 OpenGLES_Ch8_4 包含了一个简单的 UtilityBillboard 类来为每个公告牌存储位置、大小和纹理坐标。

```
//
//  UtilityBillboard.h
//
//

#import <GLKit/GLKit.h>


@interface UtilityBillboard : NSObject
@property (assign, nonatomic, readonly)
   GLKVector3 position;
@property (assign, nonatomic, readonly)
   GLKVector2 minTextureCoords;
@property (assign, nonatomic, readonly)
   GLKVector2 maxTextureCoords;
@property (assign, nonatomic, readonly)
   GLKVector2 size;
@property (assign, nonatomic, readonly)
   GLfloat signedDistanceInLookDirection;
```

```
- (id)initWithPosition:(GLKVector3)aPosition
   size:(GLKVector2)aSize
   minTextureCoords:(GLKVector2)minCoords
   maxTextureCoords:(GLKVector2)maxCoords;

- (void)updateWithEyePosition:(GLKVector3)eyePosition
   lookDirection:(GLKVector3)lookDirection;

@end
```

UtilityBillboard 实现了“-updateWidthEyePosition:lookDirection:”方法来使用 GLKVector3-DotProduct() 函数计算公告牌与眼睛位置之间的一个有符号的距离。这个有符号的距离用于从远到近对比排序公告牌。当公告牌的位置在观察者的后面时这个有符号的距离就是负值。利用这一点可以实现一个简单的优化：给定的一个根据距离排序的公告牌数组，当第一次碰到距离是负值的公告牌时就可以安全地停止绘制更多的公告牌。数组中剩下的公告牌都是不可见的，因为它们都在观察者的后面。

```
//////////////////////////////////////////////////////////////////
// Calculate the current distance of the receiver from
// eyePosition to enable sorting for render order.
- (void)updateWithEyePosition:(GLKVector3)eyePosition
   lookDirection:(GLKVector3)lookDirection;
{
   const GLKVector3 vectorFromEye = GLKVector3Subtract(
      eyePosition, position_);
   signedDistanceInLookDirection_ =
      GLKVector3DotProduct(vectorFromEye, lookDirection);
}
```

UtilityBillboardManager 类跟踪公告牌并渲染它们的方式与第 7 章的 UtilityModel-Manager 类跟踪并渲染模型的方式大致相同。通过调用“-addBillboard:”方法来添加要管理的公告牌。在绘制公告牌之前要调用 UtilityBillboardManager 类的“-updateWithEye Position:lookDirection:”方法。

UtilityBillboardManager 会更新其管理的所有公告牌并按从远到近的顺序排序这些公告牌。UtilityBillboardManager 的“-drawWithEyePosition:lookDirection:upVector:”方法会使用圆柱形或者球形数学算法来重新计算每个公告牌中的每个三角形的顶点，具体使用哪种数学算法取决于 UtilityBillboardManager 的 shouldRenderSpherical 属性，然后会使用 glDrawArrays() 函数来绘制所有的可视公告牌。

```
//
//  UtilityBillboardManager.h
//
//

#import <GLKit/GLKit.h>
```

```
@class UtilityBillboard;


@interface UtilityBillboardManager : NSObject

@property (strong, nonatomic, readonly)
   NSArray *sortedBillboards;
@property (assign, nonatomic, readwrite)
   BOOL shouldRenderSpherical;

- (void)updateWithEyePosition:(GLKVector3)eyePosition
   lookDirection:(GLKVector3)lookDirection;

- (void)addBillboard:(UtilityBillboard *)aBillboard;

- (void)addBillboardAtPosition:(GLKVector3)aPosition
   size:(GLKVector2)aSize
   minTextureCoords:(GLKVector2)minCoords
   maxTextureCoords:(GLKVector2)maxCoords;


@end

@interface UtilityBillboardManager (viewAdditions)

- (void)drawWithEyePosition:(GLKVector3)eyePosition
   lookDirection:(GLKVector3)lookDirection
   upVector:(GLKVector3)upVector;

@end
```

排序可以利用 Cocoa Touch 的内置功能。NSArray 类提供了所需的“–sortedArrayUsing-Function:context:”方法。UtilityBillboard 提供了一个叫做 UtilityCompareBillboard-Distance() 的适合与“–sortedArrayUsingFunction:context:”一起使用的函数。

```
named UtilityCompareBillboardDistance() suitable for use with
-sortedArrayUsingFunction:context:.
   ////////////////////////////////////////////////////////////////////
//
- (void)updateWithEyePosition:(GLKVector3)eyePosition
   lookDirection:(GLKVector3)lookDirection;
{
   // Make sure lookDirection is a unit vector
   lookDirection = GLKVector3Normalize(lookDirection);

   for(UtilityBillboard *currentBillboard in
      self.sortedBillboards)
```

```
    {
        [currentBillboard updateWithEyePosition:eyePosition
            lookDirection:lookDirection];
    }

    // Sort from furthest to nearest and billboards behind the
    // viewer ordered after all others.
    [self.mutableSortedBillboards
        sortUsingFunction:UtilityCompareBillboardDistance
        context:NULL];
}
```

浏览一下 UtilityBillboardManager.m 中的 UtilityBillboardManager 类的实现。这个类的代码并不多并且重用了很多在前面的类似 UtilityMesh 的类中使用的用于排序和管理顶点属性的技术。公告牌是组织网格和模型背后的几何图形的最普遍方式之一。但是，由于需要不断地更新顶点位置，公告牌会消耗大量的 CPU 和 GPU 之间宝贵的内存带宽。在 GPU 硬件中创建和实现的点精灵可以减少对公告牌的需求。

8.5 小结

特效会在尽量不增加 GPU 处理负载的情况下增加场景的视觉丰富性。每个特效都有自己的局限性。天空盒必须要放置在正确的位置范围内，以便视点不会过于扭曲纹理以至于损坏最终的渲染图像。使用点精灵实现的粒子效果是非常高效的，但是 GPU 无法从远到近排序粒子，而不排序的话，有些半透明的粒子在渲染时就会出现瑕疵。公告牌会增加场景细节，但是与天空盒类似，它也不能适用于所有情况。可以通过约束视点来避免暴露公告牌的深度缺失。

对于嵌入式图形应用来说，性能是至关重要的，并且特效通常是在保持良好性能的情况下提供一些令人印象深刻的结果。下一章会讲解优化技术，就是在特效不够时提高性能。

第9章 优 化

本章会学习能够提高 iOS 设备的 OpenGL ES 2.0 渲染性能的优化策略。在 9.1 节介绍的第一个技术通常最有效。接下来的每个技术的效果会越来越小，但是在某些情况下，使用与不使用它们仍然可能会有“足够快”和“太慢”的巨大差别。

对于嵌入式设备来说，优化不仅仅是指优化渲染的速度。例如，用户可能宁愿只要 30Hz 的刷新率而延长电池的使用时间，也不要 60Hz 的刷新率而导致设备电池快速耗尽。为每个应用考虑一个合理的平衡。最重要的是不要渲染对用户没用的任何东西。没有必要重新渲染一个没有变化的场景。也没有必要在 iOS 设备上让渲染速度超过 60Hz，因为用户永远看不到这些额外的更新。

在优化之前，请考虑一下这个优化是否真的有必要。有很多不优化的理由：

- 优化一个应用常常带来新的错误。
- 优化一个应用的代码常常让这些代码难以阅读和维护。
- 可能花了大量的时间来优化代码，但是却获得了很少的性能提升。

即使存在优化软件的机会，明智的程序员也会保持谨慎。正如算法分析之父 Donald Knuth 在 1974 年所写的：

我们应该忽视那些小的性能提升机会，大约 97% 的情况下：过早优化都是一切不幸的根源。但我们不应该放弃在剩下的关键的 3% 的情况中的性能提升机会。一个好的程序员不会因为这样的理由而放任不管，他还是会明智地仔细查看关键代码，但只会在这个代码已经被验证后。

对于 Knuth 的忠告，那 3% 的性能关键代码可能在 OpenGL ES 库中、驱动中，或者其他超出你的控制范围的地方。很多应用天生受限于输入 / 输出（I/O）速度或者其他的无法通过修改源代码来改善的因素。类似的，代码执行速度的变化对于受限于 GPU 的应用是没用的。当瓶颈不在你的代码中时，最好的方式是避免瓶颈：减少 I/O、渲染更少的几何对象，或者减少内存访问。

9.1 尽可能减少渲染

GPU 能够在 33 毫秒内完成惊人的计算量，以实现 30Hz 的渲染刷新率。尽管如此，

GPU 在给定的时间内也只能渲染这么多的点、线段和三角形。当几何图元的数量太多，超过了 GPU 的处理能力时，再怎么优化也无法获得想要的刷新率。本节会讲解一个叫做剔除（culling）的技术，使用这个技术可以在不减少输出质量的情况下，尽量减少需要提交给 GPU 的几何图元的数量。

剔除涵盖了能够避免处理对于颜色渲染缓存中的片元没有影响的几何对象的各种技术。通常剔除能够非常明显地减少 GPU 的工作量。可以不实现其他的优化，但是这个优化最好是实现了。

OpenGL ES 包含了一些内置的剔除能力。例如第 7 章就利用了“背面剔除”技术。每个三角形都有前面和后面，这个前面和后面是根据 OpenGL ES 接收三角形顶点的顺序来确定的。背面剔除就是让 OpenGL ES 丢弃那些背离观察者的三角形。在使用实心模型对象的应用中，这个优化几乎是无损的，因为用户通常永远看不到背面的三角形。

最有效的优化方式之一是避免渲染在当前视平截体外的对象。很多场景经常会包含用户永远都看不到的几何对象。只有在当前视平截体内的对象会影响在帧缓存的像素颜色渲染缓存中的片元颜色。第 2 章介绍了渲染缓存。第 5 章和第 8 章讲解了视平截体。

9.1.1 基于视平截体的剔除

之前的例子 OpenGLES_Ch8_4 使用了一个基于视平截体剔除的简单形式：使用一个简单的距离计算来判断一个点或者一个公告牌的位置在观察者的前面还是后面。这个例子会避免渲染处在观察者后面的公告牌几何图形。这个方法不仅适用于公告牌，还适用于更多的几何图形对象。假设视点处在一个复杂场景的中心，那么可能一半的几何图形对象会处在这个视点的后面，并且位于视平截体的外面。这个简单的计算机距离计算可能会从渲染列表中剔除这个场景内一半的几何图形。

稍微复杂些的计算甚至可以剔除更多。图 9-1 显示了物体相对于视平截体的位置。一个对象可能完全处在平截体内，也可能部分处在平截体内，或者完全处在平截体外部。用来确定对象完全处在平截体外部的代码，几乎总是比向 GPU 提交物体的几何图形，并避免处理没有机会影响渲染场景外观的对象的代码执行得更快。

在图 9-1 中，宇宙飞船完全处在视平截体的外部。一个太远而看不到，另一个在视点的后面，第三个在视域外部的一侧。与此相反，每个圆柱体都是完全或者部分在视平截体内部的。

例子 OpenGLES_Ch9_1 包含分散在整个场景中的 81 个宇宙飞船模型，并使用视平截体剔除技术有效地减少了 GPU 需要处理的几何图形的数量。在这个例子中的用户界面开关可以通过开启和关闭剔除来展示其影响。在第一代 iPad 上，开启剔除后，场景的刷新率是每秒 30 帧。关闭剔除后，刷新率会降到大约每秒 7 帧或者更少。在这个例子中剔除是非常有效的，因为任何时刻在视域中只有 5 到 6 个宇宙飞船。剔除可以把每帧中 GPU 需要处理的模型的数量从 81 个减少到 5 到 6 个。

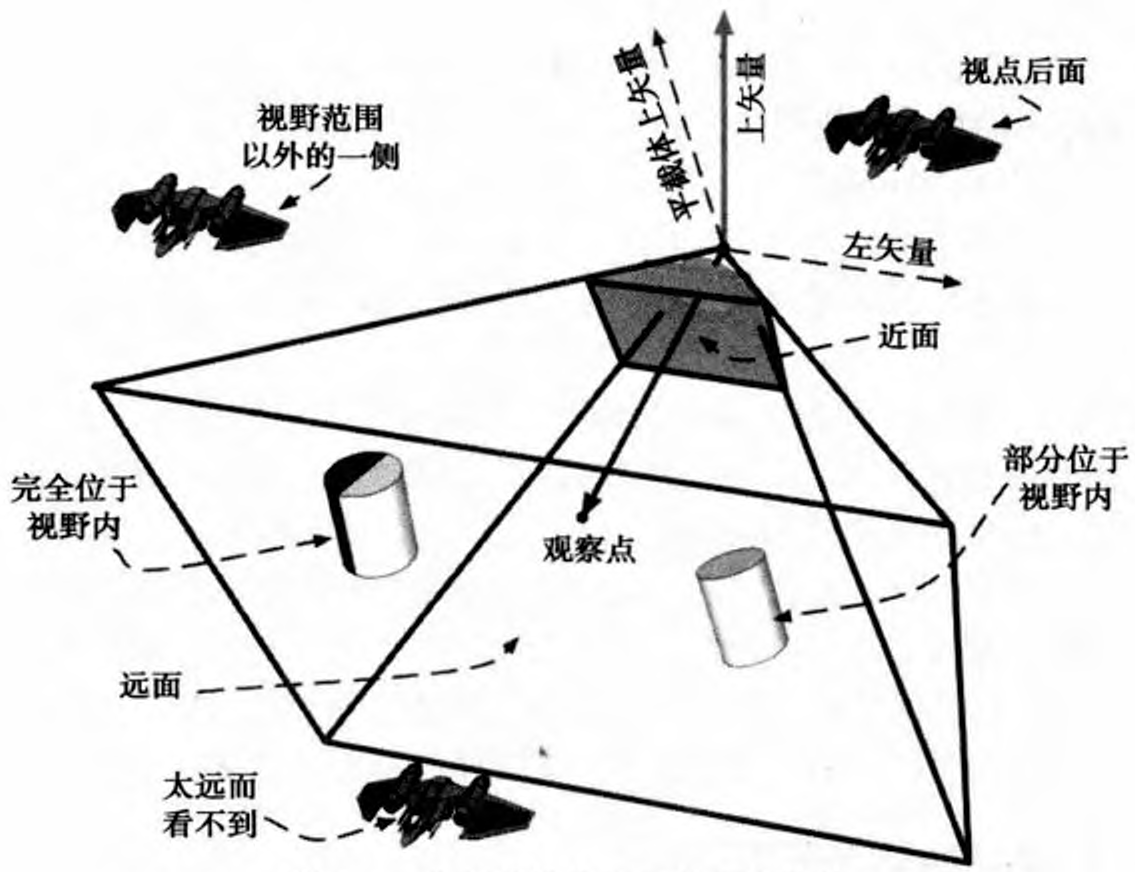


图 9-1　在平截体内部和外部的物体

在下面的 AGLKFrustum 数据类型中粗体突出的元素定义了类似图 9-2 中显示的视平截体。其他的元素源于视平截体的定义，用于帮助确定几何对象是否与视平截体相交。

```
//////////////////////////////////////////////////////////////////
// This data type is used to store the parameters that define a
// viewing frustum
typedef struct
{  // Frustum definition
   GLKVector3 eyePosition;
   GLKVector3 xNormalVector;
   GLKVector3 yNormalVector;
   GLKVector3 zNormalVector;
   GLfloat aspectRatio;
   GLfloat nearDistance;
   GLfloat farDistance;

   // Derived frustum properties
   GLfloat nearWidth;
   GLfloat nearHeight;
   GLfloat tangentOfHalfFieldOfView;
   GLfloat sphereFactorX;
   GLfloat sphereFactorY;
}
AGLKFrustum;
```

三个法向量、两个距离、一个纵横比和一个视野共同确定了视平截体，参见图9-2。不巧的是，这个平截体数据结构存储的信息与编码在投影矩阵和 model-view 矩阵中的信息是相同的。平截体数据结构只是另一种表示形式，用于测试几何图形是否与视平截体相交。可以从已初始化的平截体中提取出等价的投影和 model-view 矩阵。

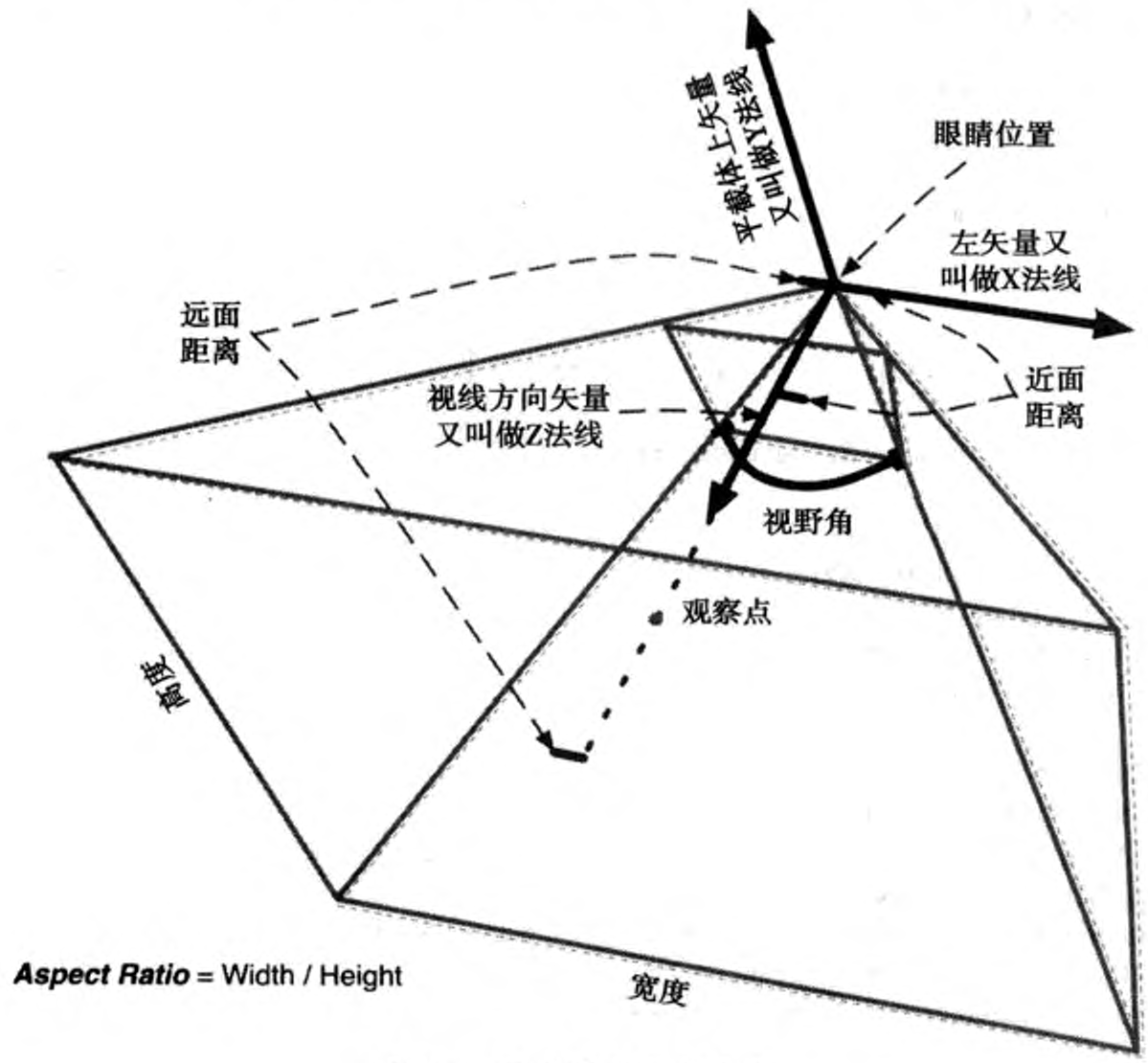


图 9-2 平截体的几何定义

下面的 AGLKFrustumMakeFrustumWithParameters(GLfloat fieldOfViewRad, GLfloat aspectRatio, GLfloat nearDistance, GLfloat farDistance) 函数会接收与 GLKMatrix4MakePerspective() 函数相同的参数并返回一个已初始化的 AGLKFrustum 数据结构。相关的 GLKMatrix4 AGLKFrustumMakePerspective(const UtilityFrustum *frustumPtr) 函数可以从一个平截体中提取出一个投影矩阵。当用来初始化平截体的参数相同时，这个函数返回的矩阵与 GLKMatrix4MakePerspective() 函数返回的矩阵是相同的。

```
AGLKFrustum AGLKFrustumMakeFrustumWithParameters
(
   GLfloat fieldOfViewRad,
   GLfloat aspectRatio,
   GLfloat nearDistance,
```

```
    GLfloat farDistance)
{
    AGLKFrustum frustum;

    AGLKFrustumSetPerspective(
       &frustum,
       fieldOfViewRad,
       aspectRatio,
       nearDistance,
       farDistance);

    return frustum;
}


////////////////////////////////////////////////////////////////////////
//
extern void AGLKFrustumSetPerspective
(
    AGLKFrustum *frustumPtr,
    GLfloat fieldOfViewRad,
    GLfloat aspectRatio,
    GLfloat nearDistance,
    GLfloat farDistance)
{
    NSCAssert(NULL != frustumPtr,
       @"Invalid frustumPtr parameter");
    NSCAssert(0.0f < fieldOfViewRad && M_PI > fieldOfViewRad,
       @"Invalid fieldOfViewRad");
    NSCAssert(0.0f < aspectRatio, @"Invalid aspectRatio");
    NSCAssert(0.0f < nearDistance, @"Invalid nearDistance");
    NSCAssert(nearDistance < farDistance, @"Invalid farDistance");

    const GLfloat halfFieldOfViewRad = 0.5f * fieldOfViewRad;

    // store the information
    frustumPtr->aspectRatio = aspectRatio;
    frustumPtr->nearDistance = nearDistance;
    frustumPtr->farDistance = farDistance;

    // compute width and height of the near section
    frustumPtr->tangentOfHalfFieldOfView =
       tanf(halfFieldOfViewRad);
    frustumPtr->nearHeight = nearDistance *
       frustumPtr->tangentOfHalfFieldOfView;
    frustumPtr->nearWidth = frustumPtr->nearHeight * aspectRatio;

    // Calculate sphere factors (used when testing sphere
    // intersection with frustum)
```

```
    frustumPtr->sphereFactorY =
       1.0f/cosf(frustumPtr->tangentOfHalfFieldOfView);
    const GLfloat angleX =
       atanf(frustumPtr->tangentOfHalfFieldOfView * aspectRatio);
    frustumPtr->sphereFactorX = 1.0f/cosf(angleX);

}
```

除了可以存储透视信息，视平截体还可以存储用于定义渲染场景的视点的值。下面的 AGLKFrustumSetPositionAndDirection (AGLKFrustum *frustumPtr, GLKVector3 eyePosition, GLKVector3 lookAtPosition, GLKVector3 upVector) 函数可以使用与 GLKMatrix4-MakeLookAt() 函数类似的参数来更新平截体的视点。可以使用相关的 GLKMatrix4AGLKFrustumMakeModelview(const AGLKFrustum *frustumPtr) 函数来获得与平截体的视点对应的 model-view 矩阵，当用设置平截体位置和方向的参数调用 GLKMatrix4MakeLookAt() 函数时，求得的 model-view 矩阵与这个函数返回的矩阵是相同的。

```
void AGLKFrustumSetPositionAndDirection
(
   AGLKFrustum *frustumPtr,
   GLKVector3 eyePosition,
   GLKVector3 lookAtPosition,
   GLKVector3 upVector)
{
   NSCAssert(NULL != frustumPtr,
      @"Invalid frustumPtr parameter");

   frustumPtr->eyePosition = eyePosition;

   // compute the Z normal of the frustum. The Z normal points in
   // the direction from eye position to look at position
   const GLKVector3 lookAtVector =
      GLKVector3Subtract(eyePosition, lookAtPosition);
   NSCAssert(0.0f < AGLKVector3LengthSquared(lookAtVector),
             @"Invalid eyeLookPosition parameter");
      frustumPtr->zNormalVector = GLKVector3Normalize(lookAtVector);
  // The frustum's X normal is the cross product of the
  // normalized "up" vector and the frustum's Z normal
  frustumPtr->xNormalVector = GLKVector3CrossProduct(
     GLKVector3Normalize(upVector),
     frustumPtr->zNormalVector);

  // The frustum's Y normal is the cross product of the
  // frustum's Z normal and the frustum's X normal.
  frustumPtr->yNormalVector = GLKVector3CrossProduct(
     frustumPtr->zNormalVector,
     frustumPtr->xNormalVector);
}
```

为了测试一个点是否落在一个平截体内部，第一步是计算从眼睛的位置到当前测试的点之间的矢量，然后提取出这个矢量的平截体轴对齐分量。回顾一下 1.4 节，每个 3D 矢量都可以表示为三个不同的轴对齐分量矢量的结合。图 9-3 显示了与每个平截体轴对齐的分量矢量。分量矢量的总和相当于从眼睛的位置到当前测试的点之间的矢量。

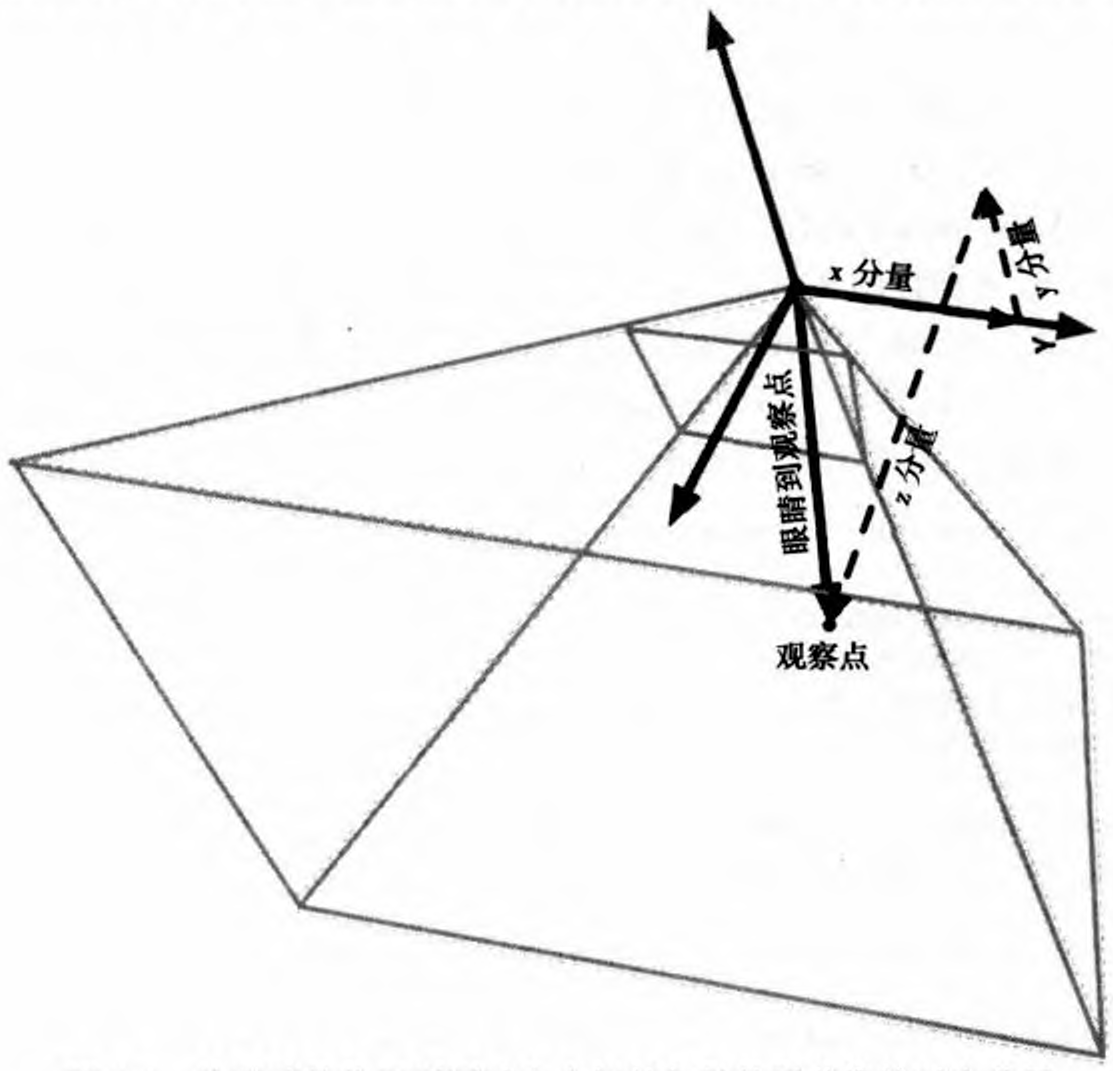


图 9-3　从眼睛的位置到测试点之间的矢量的平截体轴对齐分量

分量矢量的长度是使用“眼睛到点”的矢量与平截体的每个标准化轴矢量之间的标量积计算出来的。第一个也是最快的测试是确定这个点是否在平截体远面的前面或者在平截体近面的后面。如果第一个测试的结果为否，那么就没有必要再做任何工作了。下面摘录自例子 OpenGLES_Ch9_1 的 AGLKFrustumComparePoint（const AGLKFrustum *frustumPtr，GLKVector3 point）函数的代码会计算从眼睛位置到测试点之间的矢量。然后提取了这个矢量与平截体 Z 轴对齐的分量。如果这个 Z 轴分量要比远面大，或者比近面小，那么这个点就处在平截体的外部。

```
// compute vector from camera position to point
const GLKVector3 eyeToPoint = GLKVector3Subtract(
   frustumPtr->eyePosition, point);

// compute and test Z coordinate within frustum
const GLfloat pointZComponent = GLKVector3DotProduct(
```

```
    eyeToPoint, frustumPtr->zNormalVector);

if(pointZComponent > frustumPtr->farDistance ||
    pointZComponent < frustumPtr->nearDistance)
{  // The point is not within frustum
        result = AGLKFrustumOut;
}
```

下一步，这个眼睛到点的矢量的 Y 分量必须要短于在被测点的 Z 位置处的平截体的高度。如图 9-4 所示，这个高度是通过用视野角度的一半的正切值乘以被测点的 Z 位置计算出来的。

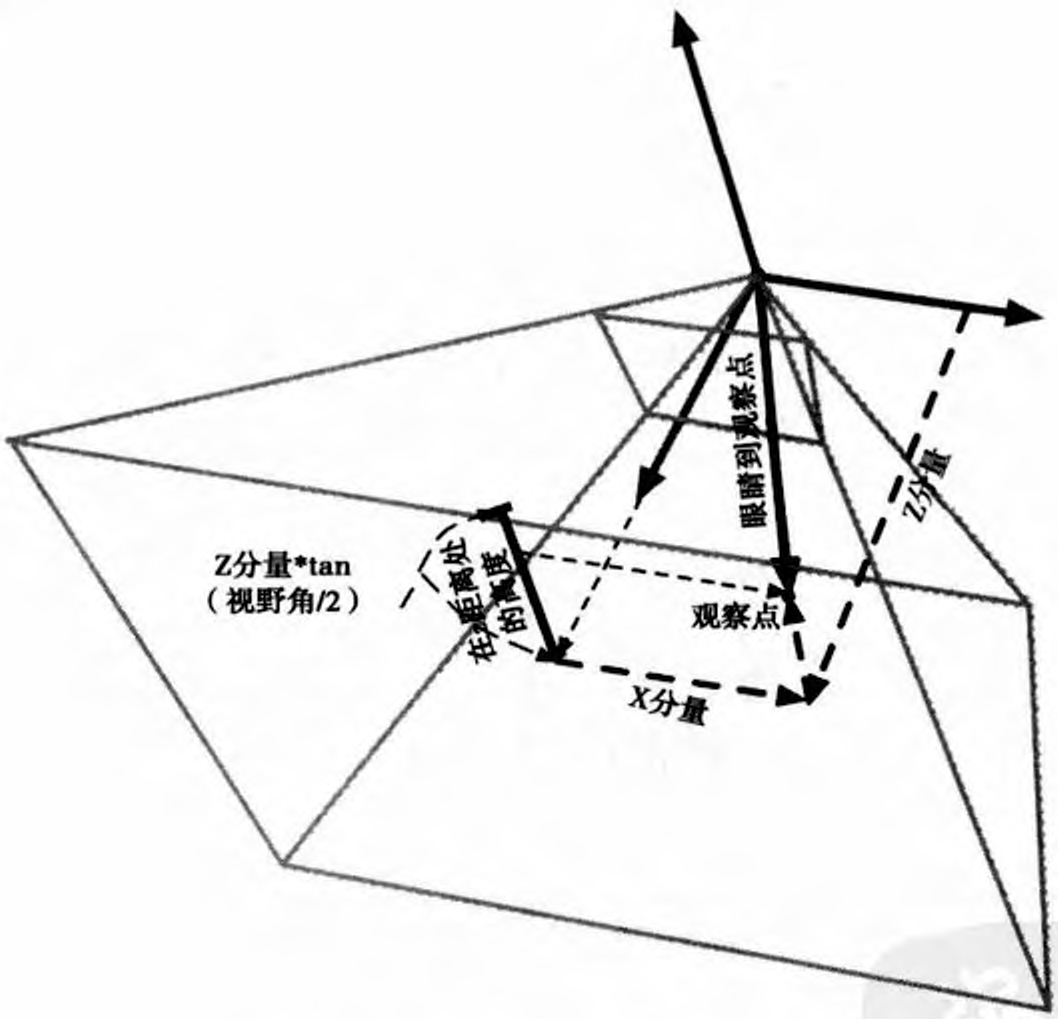


图 9-4 在 Z 位置处的平截体高度

最后，这个眼睛到点的矢量的 X 分量必须要短于在被测点的 Z 位置处的平截体的宽度。平截体的这个宽度等于平截体的高度乘以纵横比。下面的摘录代码会测试 X 和 Y 分量是否在平截体内部。

```
// compute and test Y coordinate within frustum
const GLfloat pointYComponent =
    GLKVector3DotProduct(eyeToPoint,
        frustumPtr->yNormalVector);
const GLfloat frustumHeightAtZ = pointZComponent *
    frustumPtr->tangentOfHalfFieldOfView;
```

```
if(pointYComponent > frustumHeightAtZ ||
   pointYComponent < -frustumHeightAtZ)
{  // The point is not within frustum
   result = AGLKFrustumOut;
}
else
{  // compute and test the X coordinate within frustum
   const GLfloat pointXComponent =
      GLKVector3DotProduct(eyeToPoint,
         frustumPtr->xNormalVector);
   const GLfloat frustumWidthAtZ = frustumHeightAtZ *
      frustumPtr->aspectRatio;

   if(pointXComponent > frustumWidthAtZ ||
      pointXComponent < -frustumWidthAtZ)
   {  // The point is not within frustum
      result = AGLKFrustumOut;
   }
}
```

测试一个球体是否与视平截体相交会更复杂一点。好像测试从球体的中心到平截体内的一个点之间的距离小于球体的半径几乎就足够了。不幸的是，这个测试漏掉了一个特殊情况，参见图 9-5。

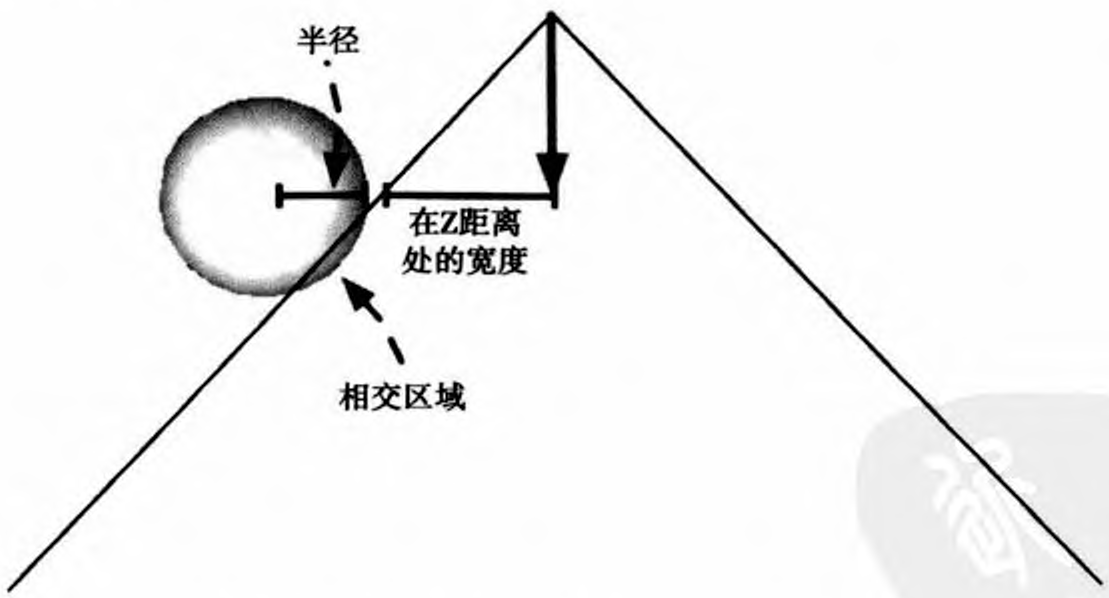


图 9-5　测试球体与平截体相交的一个特殊情况

例子 OpenGLES_Ch9_1 中的 AGLKFrustum 数据类型包含了命名为 sphereFactorX 和 sphereFactorY 的专门用于处理球体的特殊情况的元素。这些因数的值稍大于 1.0，当测试球体与平截体相交时，会用这些因数乘以球体的半径。图 9-6 显示了球体因数与半径之间的几何关系。

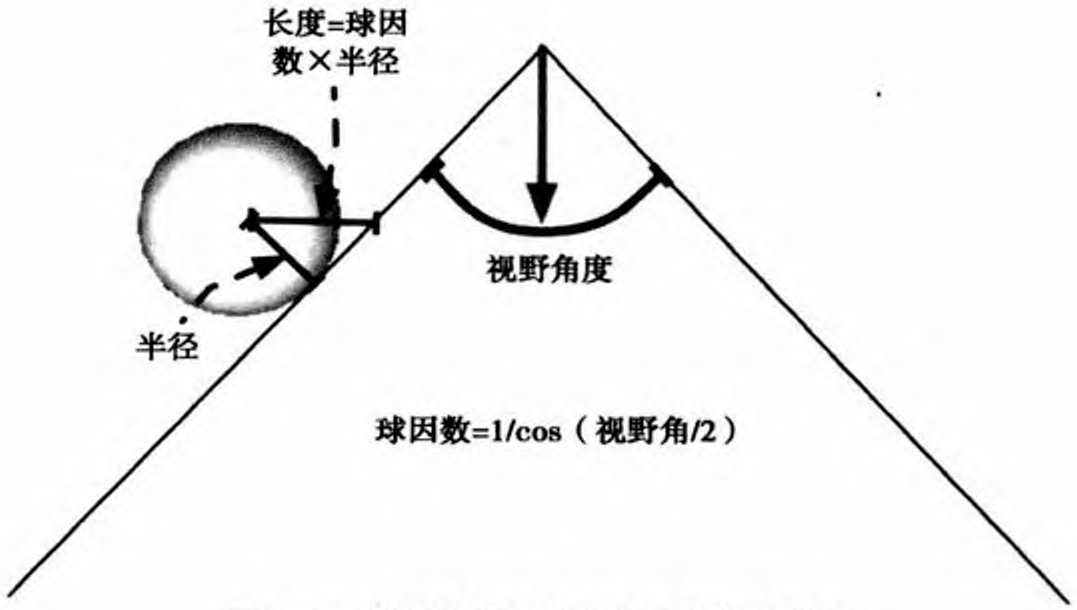


图 9-6 球体因数和半径之间的几何关系

球体因数是基于球体的视野角度计算出来的，参见下面的代码：

```
// Calculate sphere factors (used when testing sphere
// intersection with frustum)
frustumPtr->sphereFactorY =
   1.0f/cosf(frustumPtr->tangentOfHalfFieldOfView);
const GLfloat angleX =
   atanf(frustumPtr->tangentOfHalfFieldOfView * aspectRatio);
frustumPtr->sphereFactorX = 1.0f/cosf(angleX);
```

所有的几何关系都是为了有效地测试球体是否与视平截体相交。仔细检查例子 OpenGLES_Ch9_1 中的 AGLKFrustumCompareSphere (const AGLKFrustum *frustumPtr, GLKVector3 center, float radius) 函数的实现以感受一下实际的应用。要确定一个模型是否与平截体相交，只要测试包裹这个模型的球体是否与这个平截体相交就足够了。例子中的“-drawModels”方法试图用第 7 章和第 8 章例子中的相似代码来绘制散落在整个场景中的 81 个模型。如下代码中粗体显示的是一个快速测试，这个测试会剔除在当前渲染帧中不可见的模型。

```
//////////////////////////////////////////////////////////////////
// Draw models positioned throughout the scene
- (void)drawModels
{
   const float modelRadius = 7.33f; // Used to cull models

   self.baseEffect.texture2d0.name =
      self.modelManager.textureInfo.name;
   self.baseEffect.texture2d0.target =
      self.modelManager.textureInfo.target;

   [self.modelManager prepareToDraw];
```

```
// Draw an arbitrary large number of models
for(NSInteger i = -4; i < 5; i++)
{
   for(NSInteger j = -4; j < 5; j++)
   {
      const GLKVector3 modelPosition = {
         -100.0f + 150.0f * i,
         0.0f,
         -100.0f + 150.0f * j
      };

      if(!self.shouldCull ||
         AGLKFrustumOut != AGLKFrustumCompareSphere(
            &frustum_, modelPosition, modelRadius))
      {
         // Save the current matrix
         GLKMatrix4 savedMatrix =
            self.baseEffect.transform.modelviewMatrix;

         // Translate to the model position
         self.baseEffect.transform.modelviewMatrix =
            GLKMatrix4Translate(savedMatrix,
               modelPosition.x,
               modelPosition.y,
               modelPosition.z);
         [self.baseEffect prepareToDraw];

         // Draw the model
         [self.model draw];

         // Restore the saved matrix
         self.baseEffect.transform.modelviewMatrix =
            savedMatrix;
      }
   }
}
```

为了保持例子 OpenGLES_Ch9_1 的简单性，每次渲染场景时都会测试每个模型与平截体是否相交。通过把模型位置存储在一个叫做场景图的数据结构中可以避免很多的测试。

场景图辅助剔除

场景图（scene graph）是一个使用树形数据结构的几何对象层次组织。在这个树形结构中有一个根元素。在这个数据结构中，这个根元素在概念上是其包含的其他元素的“父”。每个根元素的“子”也可能是其他元素的“父”。在 Cocoa Touch 应用中的

视图层次结构就是 2D UIView 实例的场景图，参见苹果的帮助文档：http://developer.apple.com/library/ios/documentation/General/Conceptual/Devpedia-CocoaApp/View%20Hierarchy.html。同样的概念也适用于 3D 对象的层次结构。由于“父”元素在空间上包含它的所有子元素，因此只测试父元素是否与平截体相交就足够了。如果父元素在平截体外部，那么根据定义所有它的子元素也在平截体外部，因此没有必要单独地测试每个子元素。

维基百科上有关于场景图的一个详细解释，地址为：http://en.wikipedia.org/wiki/Scene_graph。高效图形应用通常使用一个叫做八叉树（octree）的相关数据结构，详见 http://en.wikipedia.org/wiki/Octree。本书中没有一个例子有足够的对象以受益于完全基于八叉树的场景图。但是，在第 10 章中的例子 OpenGLES_Ch10_1 在绘制一个由丘陵和山谷组成的模拟地形时，会使用一个简单形式的用于剔除的空间优化。地形本身是一个类似第 6 章中的网格的大网格。

例子 OpenGLES_Ch10_1 的地形网格包含的顶点超过了一次 glDrawElements() 函数调用能够访问的最大量。因此，需要把这个地形网格分割成“瓦片（tiles）”，每个瓦片引用这些顶点的一个子集。瓦片足够小，因而可以使用 glDrawElements() 函数来绘制，多个瓦片共同组成一个简单的场景图。模型和其他的几何对象成为位于瓦片内的子元素。如果一个瓦片没有与平截体相交，那么瓦片内的对象也是不可见的。

9.1.2 简化

如果你无法避免绘制一些东西，那就让这个绘制保持简单。使用类似公告牌的特效可以减少传给 GPU 的几何对象的复杂性。减少网格中的顶点数量。在保证渲染质量的情况下使用尽可能小的纹理，考虑使用第 3 章介绍的压缩纹理。使用第 3 章介绍的多重纹理来避免多通道渲染。如果可能的话，多重纹理也是应该避免的。使用在第 4 章介绍的烘焙灯光来避免 GPU 的灯光计算。灯光需要 GPU 执行一些计算代价很高的操作。下一章的例子 OpenGLES_Ch10_1 会为地形网格使用灯光预计算以减轻 GPU 的负担。

当你花时间做优化时，剔除和简化总是能产生最好的回报。当渲染对象时，再怎么提高效率也不如渲染更少的对象节省的时间多。再怎么优化灯光效果也不如完全不做灯光计算节省的时间多。

9.2 不要猜：解析（Profile）

一般情况下，一个程序的执行时间有 80% 是耗费在 20% 的代码上的。在非常罕见的情况下，一个应用的每个代码块中所花费的时间是相同的，当出现这种情况时，优化量会变得很大。在这种情况下，必须要优化所有的代码块才能获得显著的性能改善。幸运的是，对于大部分应用来说，在找出了这关键的 20% 的代码后，仅仅对这 20% 的代

码做重点优化就可以用最少的努力获得最大的回报。甚至只优化 20% 的代码可能也太多。Knuth 建议集中在最能决定性能的那 3% 的代码。关键是要找出决定性的代码。

苹果的 Xcode 集成开发环境包含了一些用于找出性能瓶颈的工具。性能可能受限于 CPU 或 GPU 或者强制一个处理程序等待另一个的同步事件。但是，在你使用强有力的分析工具之前，有一个快速简单的缩小性能问题的搜索范围的方法：注释掉应用的所有绘图代码并测试之后的“刷新率”。屏幕内容实际上不会变化，因为什么都没有绘制，如果这个应用甚至都不能按照所需的速度调用注释掉的代码，再怎么对绘图做优化都不能解决问题。在这种情况下，性能瓶颈不在绘图代码中。

当 OpenGL ES 渲染确实限制了应用性能时，可以使用苹果的 OpenGL ES 性能检测工具来找出所有 GPU 瓶颈。在很多情况下，这些工具能够直接为你指出检测到的应用代码，如果有的话，这就是限制 OpenGL ES 性能的代码。这些工具甚至能够生成性能改进建议。

9.2.1 工具 OpenGL ES Performance Detective

OpenGL ES Performance Detective 运行在你开发用 Mac 上。为了使用它，请确保要分析的应用已安装在 iOS 设备上，并且已与电脑连接。使用 Xcode 启动 OpenGL ES Performance Detective，即在 Open Developer Tool 的子菜单中点击 OpenGL ES Performance Detective 菜单项。OpenGL ES Performance Detective 会显示一个对话框让你选择连接哪一个设备以及分析哪一个应用。在 iOS 应用开始运行后，操作你的 iOS 应用直到到达你想测试的状态，然后点击 OpenGL ES Performance Detective 的 Collect Evidence 按钮。OpenGL ES Performance Detective 会生成一个报告。

例如，当在例子 OpenGLES_Ch9_1 中开启平截体剔除后，OpenGL ES Performance Detective 会生成下面的报告：

```
Summary
The frame rate of your app is not limited by the graphics pipeline.

Your performance is not limited by the OpenGL ES commands issued by
your app. Use the Instruments tool to investigate where your
application is bottlenecked.
```

例子 OpenGLES_Ch9_1 每秒可以稳定生成 30 帧，即 GLKViewController 的 preferredFramesPerSecond 属性的默认值。每次 GPU 在开始渲染下一帧之前会完成渲染当前帧。OpenGL ES Performance Detective 没有报告的是完成每帧的渲染后还剩下多少时间。通过设置视图控制器的 preferredFramesPerSecond 属性为 60 来试着做更快的渲染，这个过程中可能会发现新的基于你的 iOS 设备的 GPU 速度的瓶颈。

当关闭基于视平截体的剔除后，OpenGL ES Performance Detective 马上会检测到一个问题并提出解决方案：

```
Summary
The frame rate of your app is limited by the graphics pipeline.
Your performance is limited by the OpenGL ES commands issued by your app.

Top Suspect
GPU Vertex Processing

Reduce the number of vertices sent to OpenGL ES by simplifying your geometry. For
example:

1. Employ a polygon reduction algorithm.
2. Do not send unused vertex attributes. For example, do not send a normal when
lighting is disabled.
3. Use a Level of Detail (LOD) algorithm to avoid processing 1000's of triangles for
an object 10's of pixels on the screen.
4. Submit sorted indexed triangles. Sort both the index list and triangle list in
strip order.
5. Perform coarse object culling on the CPU.
```

OpenGL ES Performance Detective 的编号建议并不是按照它们应该被实施的顺序来列出的。对于例子 OpenGLES_Ch9_1 来说，按照编号 5 的建议重新开启基于视平截体的剔除就可以实现显著的性能提高。

9.2.2 工具 Instruments

苹果的 Instruments 应用运行在 Mac OS X 上，可以实现对于 iOS 应用执行时的几乎所有因素的详细分析。Instruments 应用自身会加载被称为“仪器”的各种不同工具来收集信息。你可以挑选可加载的“仪器”来改进分析。例如，Time Profiler 可以有效地收集应用代码执行时的统计数据，并帮助找出最耗费时间的代码。如果 Time Profiler 指示最耗费时间的是系统调用，那么可以使用 System Trace 来记录这个应用所做的所有系统调用。如果 Time Profiler 指示内存分配耗费的时间太多，那么可以使用 Allocations 来找出内存分配的源头。OpenGL ES Analyzer 和 OpenGL ES Driver 可以用来测量 OpenGL ES 活动。使用这些仪器逐个深入直到找到具体的瓶颈。

可以从 Xcode 4 集成开发环境的 Product 菜单的 Profile 子菜单启动 Instruments 应用。启动后，最好的方式是使用这些可用“仪器”进行测试，感受一下它们所提供的信息。也可以创建自定义的可加载“仪器”。苹果公司提供了 Instruments 的用户指南，地址为：http://developer.apple.com/library/ios/DOCUMENTATION/DeveloperTools/Conceptual/InstrumentsUserGuide/AboutTracing/AboutTracing.html。苹果公司经常更新、改进，并扩展 Instruments，这使得在线文档成为最好最新的用户指南。

9.3 尽量减少缓存复制

GPU 每秒需要访问顶点属性缓存、纹理缓存和渲染缓存数百万次。一般情况下，当 CPU 正在访问一个缓存时，GPU 就不能在这一时间访问这个缓存了。当 CPU 试图读写缓存内容时，可能会迫使 CPU 或者 GPU（一些情况下两者一起）停下来并等待。这个等待叫做卡顿（stall）。为了减少卡顿的危险，OpenGL ES 上下文和某些情况下的底层 OpenGL ES 硬件设备驱动会创建私有的缓存副本。复制缓存会耗费内存和处理器时间，但是 OpenGL ES 通常会用最少的中断时间来执行复制以避免卡顿。

尽量减少缓存复制的最好方式是初始化缓存，接着使用 GL_STATIC_DRAW 提示调用 glBufferData() 函数来把它们交给 OpenGL ES，然后就永远不要再用在 CPU 内执行的程序修改或者读取缓存了。OpenGL ES 会复制由 glBufferData() 初始化的缓存到 GPU 控制的内存，然后就可以用最佳效率来访问它们了。类似的，GPU 会向在 GPU 控制的内存中的渲染缓存写数据。试图用 CPU 通过 glReadPixels() 函数来读取渲染缓存中的内容可能会导致卡顿，并常常会迫使 OpenGL ES 耗费时间和内存来创建渲染缓存内容的副本。

当然，有时也需要动态更新缓存或者读取渲染缓存的内容。否则将无法完成应用的目的，又或者不这样做的话性能甚至会更糟。在这种情况下，推荐使用双缓存。例如：为 GPU 提供一个顶点属性缓存后，接着开始利用 CPU 处理另一个。基于当前在 GPU 中使用的缓存的所有 OpenGL ES 渲染命令都发送完后，通过调用 glBindBuffer() 函数来切换缓存，以便把 CPU 刚刚更新的缓存用于接下来的渲染，并用 CPU 更新先前 GPU 使用的缓存。只要确保 GPU 和 CPU 在同一时间不会工作在相同的缓存下即可。

苹果公司详细解释了双缓存技术，地址为：https://developer.apple.com/library/ios/documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/OpenGLESApplicationDesign/OpenGLESApplicationDesign.html。

9.4 尽量减少状态变化

OpenGL ES 上下文存储了大量的用于控制渲染运算的信息。其中一些信息缓存在 CPU 控制的内存中，另一些信息存储在 GPU 寄存器中。上下文状态的变化通常需要一些 GPU 计算，并且寄存器值变化时，可能也会拖慢 GPU 的速度。尤其要避免不必要的对于 glBindBuffer() 和 glVertexAttribPointer() 以及 glEnable() 和 glDisable() 函数的调用，具体是指避免在它们的所有变体函数中的调用，比如 glEnableVertexAttribArray()。在开启一个模式后就没有必要再重新开启了。模式不变的情况下多次调用 glEnable() 没有任何益处，而且会浪费 CPU 时间。例如，调用 glEnable(GL_DEPTH_TEST) 多次会

浪费时间来更新上下文的状态，即使该值与当前值是相同的。

iOS 4.0 和之后的 OpenGL ES 实现包含了一个新近的非标准 OpenGL ES 扩展，用来帮助合并上下文状态的变化。很多应用会在一个渲染帧中多次调用 glBindBuffer()、glEnableVertexAttribArray() 和 glVertexAttribPointer() 函数。例如，应用可能会使用不同的顶点属性来渲染多个网格。新的顶点数组对象（VAO）扩展会记录当前上下文中与顶点属性相关的状态，并存储这些信息到一个小的缓存中。后面，整个状态可以通过单次调用 glBindVertexArrayOES() 函数来恢复，而不用单独地调用 glBindBuffer()、glEnableVertexAttribArray() 和 glVertexAttribPointer() 函数。“OES”扩展是 OpenGL ES 标准的维护者（Khronos Group，Inc.）提出的一个非标准扩展。

遵循其他类型缓存的使用方式，在多个步骤中生成、绑定、初始化顶点数组对象。

```
glGenVertexArraysOES(1,                          // STEP 1
   &vertexArrayObjectID);

glBindVertexArrayOES(vertexArrayObjectID); // STEP 2

// STEP 3: initialize the bound VAO by calling
// glBindBuffer(), glEnableVertexAttribArray(), and
// glVertexAttribPointer()to configure needed vertex
// attributes
```

想要恢复一个顶点数组对象所代表的整个顶点数组属性状态，只要重新绑定 VAO 即可。OpenGL ES 函数使用这个恢复的状态，就好像已经单独地显式调用了 glBindBuffer()、glEnableVertexAttribArray() 和 glVertexAttribPointer() 函数。

```
glVertexAttribPointer() had been made.

glBindVertexArrayOES(vertexArrayObjectID);

glDrawElements(...
```

由于顶点数组对象是非标准的，因此也应该避免使用它，除非分析表明与顶点数组属性缓存相关的上下文状态变化表现出一个显著的瓶颈。VAO 可能会变成未来的 OpenGL ES 标准的正式组成部分，但是这并没有任何保证，使用 VAO 的应用编程接口（API）可能稍作修改就作为标准化进程的一部分。如果最终的标准与非标准的实现方式在很大程度上是不同的，那么使用非标准 VAO 扩展的 iOS 应用可能需要修改以使用最新的标准。

9.5 小结

iOS 设备上的 GPU 可以以一个合理的帧速率来渲染复杂的场景，但这是有限制的。

你可以做的最有效的优化就是简单地减少绘图。使用基于视平截体的剔除可以避免在用户看不到的几何图形上浪费处理器周期。解析应用的性能以找出瓶颈，然后对瓶颈做集中优化。如果一个代码块很少执行，那就不用刻意地让这个代码块的执行快 10 倍。用户永远不会注意这个加速。别忘了 iOS 设备有相对小的屏幕。小的纹理和简单的几何图形通常就可以产生适当的视觉质量。不要让几何模型太复杂以至于组成它的三角形比一个屏幕像素还小。用户永远看不到这些细节。

第 10 章会介绍更多的 3D 图形技巧，包括可以让用户与渲染对象交互的技术。如果用户感觉就好像他们正在与自己看到的对象接触并交互，那么触摸输入会变得引人入胜。

第 10 章　地形和拾取

本章会通过重用在前面章节中介绍的很多 3D 图形概念和例子类来绘制带有起伏地形的广阔户外场景。3D 地形渲染方式适用于任意的可视化大型数据集的应用，包括医疗成像和分子模拟。3D 应用中的大型数据集需要做特殊处理。有必要把数据分割成适合 GPU 访问的子集。

本章还会介绍一个叫做拾取（picking）的技术，用来检测用户与 3D 几何图形的交互。当用户触摸屏幕时，在用户手指下面的会是什么 3D 对象？触摸到的会是什么地形位置?

10.1　地形的实现

第 6 章介绍了用于有效表示复杂几何图形的网格。网格存储了共享顶点或者边的三角形集合。除了可以表示类似第 6 章中的碰碰车的模型，网格还能定义连绵起伏的丘陵和山谷以提供跨越千里的地形效果。例子 OpenGLES_Ch10_1 演示了地形网格的加载和显示。图 10-1 显示的是一个简化的地形网格。

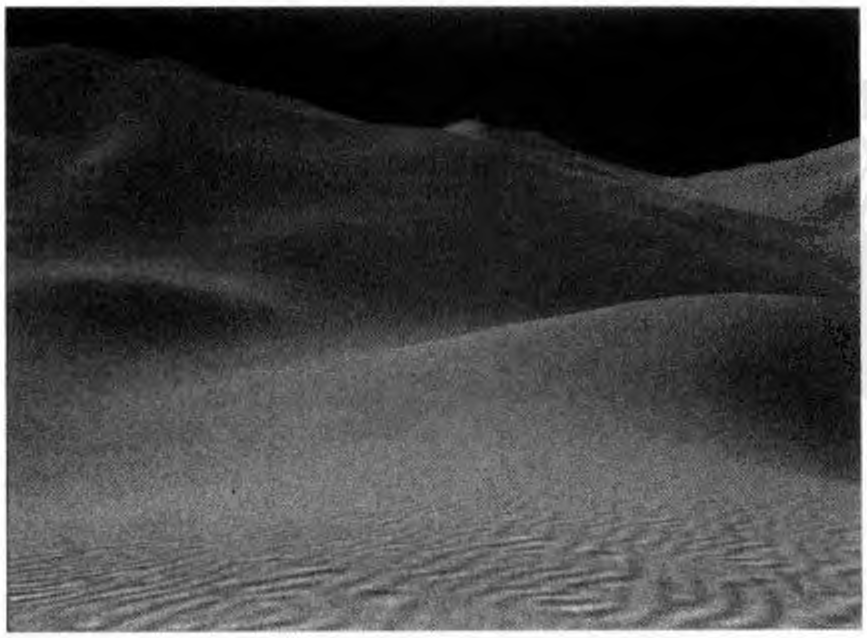


图 10-1　地形网格示例

10.1.1 高度图

地形网格会形成在坐标系的 X、Z 平面上的矩形网格。网格中每个顶点的 Y 位置对应于一个高度值（也称为“海拔”）。图 10-2 显示的是图 10-1 中的相同地形的网格线。

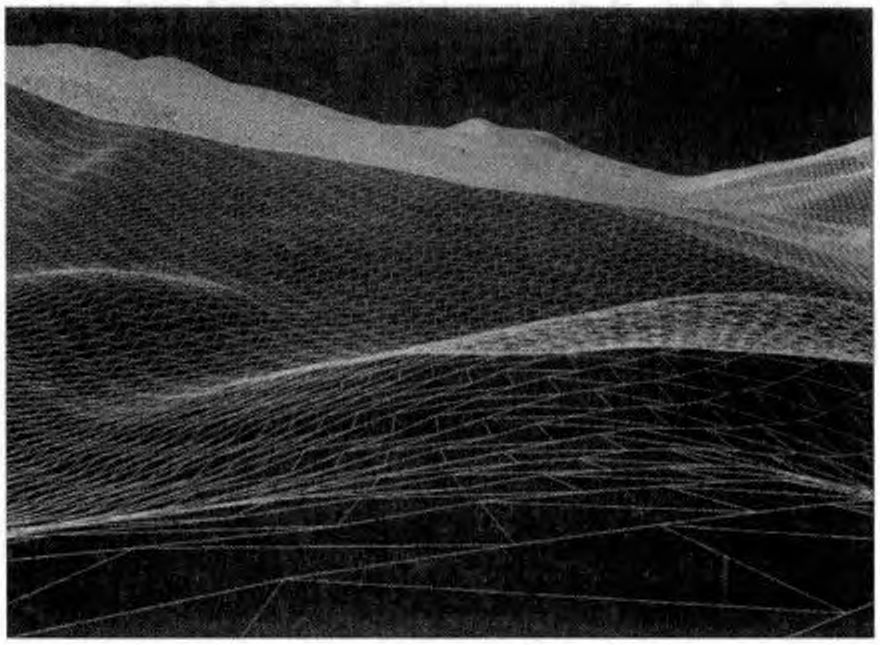


图 10-2　地形网格的网格线示例

硬编码一个复杂网格中的每个顶点的位置是不现实的。获得高度值的最简单方式是在运行时从文件中加载它们。很多 3D 建模工具都具有生成地形高度数据的功能。类似美国地质调查局（U.S. Geological Survey，网址：http://earthexplorer.usgs.gov）的网站免费提供了包含真实世界的数字高程模型（Digital Elevation Model，DEM）数据的文件。DEM 数据存在很多格式。当前的标准是多分辨率地形高程数据 2010（Multi-resolution Terrain Elevation Data 2010，GMTED2010），可以使用美国地质调查局的基于 Web 的 GMTED_viewer 来把这种标准格式转换成普通的图片文件。

例子 OpenGLES_Ch10_1 会从一个紧凑二进制文件中加载高度值，这个二进制文件是由 Mac OS X 平台上的 TerrainEditor 应用生成的，TerrainEditor 应用的源码位于：http://opengles.cosmicthump.com/learning-opengl-es-sample-code/.TerrainEditor.app，这个应用可以读取普通的图片文件，然后把这个图片中每个像素的亮度值解析为在 0.0（黑）到 1.0（白）范围内的标准化值。例子 OpenGLES_Ch10_1 会加载并缩放标准化的高度值以产生想要的地形。就实现细节来说，TerrainEditor.app 和例子 OpenGLES_Ch10_1 都是使用苹果公司的 Core Data 框架来实现紧凑的二进制表示形式的。

注意　Core Data 使用数据库概念简化了 Mac OS X 和 iOS 应用的模型数据的存储、恢复和处理。Core Data 自动化了用来管理相关联对象的常见任务，比如变更验证和内置的撤销 / 重做任务。把 Core Data 与 GLKit 和 OpenGL ES 结合起来可以提供图形较少的应用具有的相同优势。参考网页：http://developer.apple.com/technologies/ios/data-management.html。

图 10-3 显示的是用于为例子 OpenGLES_Ch10_1 提供高度值的图像。这个图像有 300 像素高，300 像素宽，代表了一个边长为 3 千米的正方形。这个图像的 90000 个像素（300×300）中的每一个像素都对应于矩形网格中的一个 X、Z 位置，同时每个像素的亮度值决定了每个位置的 Y 值。在这个网格中的每个矩形都是由两个三角形组成的。因此，要绘制整个地形必须要渲染 180000 个三角形。

图 10-3　例子 OpenGLES_Ch10_1 中的高度值的来源

10.1.2　地形瓦片

在前面的例子中，整个网格是在对 glDrawElements() 函数的一次调用中绘制的。一个网格包含 90000 个顶点，180000 个三角形，这远远超过了 OpenGL ES 的 glDrawElements() 函数的极限。因此，例子 OpenGLES_Ch10_1 会以多个叫做瓦片（tiles）的较小的组，向 glDrawElements() 函数提交网格数据。整个地形网格是用 TETerrain 类存储的，TETerrain 类是 Core Data 自动生成的，具体生成自一个在 Xcode 中图形化创建的模型，参见图 10-4。

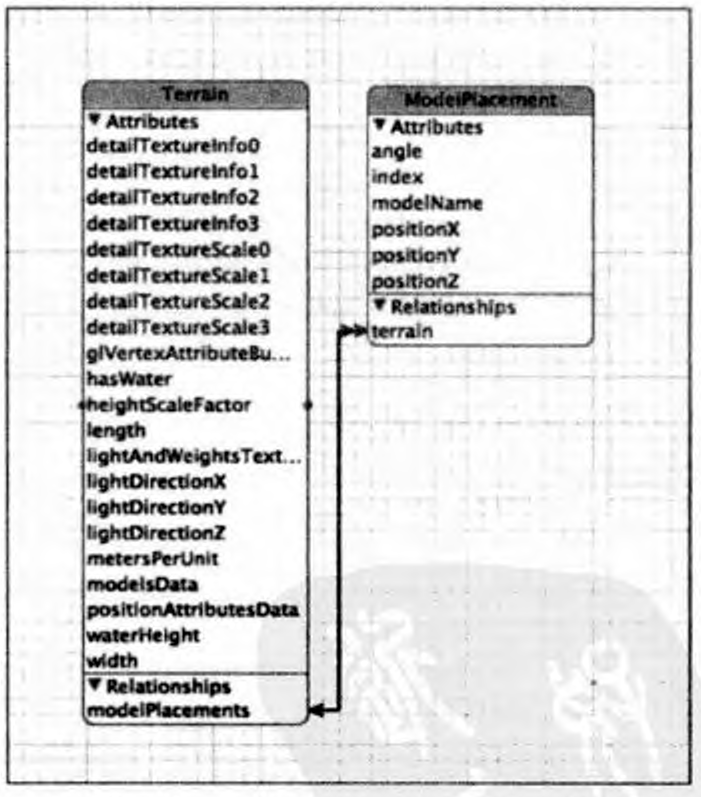


图 10-4　一个地形的 Core Data 模型

让 Xcode 生成与一个 Core Data 模型对应的 Objective-C 类并不总是必要的。但是，在例子 OpenGLES_Ch10_1 中，这样做可以方便地生成所要使用的 Objective-C 类，可以方便地添加与数据存储不相关的方法。有一个技巧：不要修改生成的代码，因为如果哪天你想修改数据模型，你可能需要重新生成它们。重新生成会清除你对前面生成的代码所做的所有编辑。不过，你可以在单独的文件中使用 Objective-C 类别向这些生成的类添加方法。

注意　Objective-C 类别用于向现存类添加方法，即使现存类的源码不可用或者需要避免对现存 / 生成的类做编辑。类别让使用多个文件来实现一个类成为可能。使用类别添加的方法与在主类实现中定义的方法在运行时是没有区别的。要了解更多信息，请访问：http://developer.apple.com/library/ios/documentation/General/Conceptual/DevPedia-CocoaCore/Category.html。

下面的代码定义了 TETerrain 类的类别。类别通常是基于被扩展的类的名字来命名的。这个例子包含了命名为 TETerrain+viewAdditions 和 TETerrain+modelAdditions 的类别。类别添加的方法会获取关于地形的信息，把整个地形分割成瓦片，并绘制瓦片。

```
//
//  TETerrain+viewAdditions.h
//
//

#import "TETerrain+modelAdditions.h"
#import <GLKit/GLKit.h>

@class GLKTextureInfo;
@class UtilityCamera;
@class UtilityTerrainEffect;
@class UtilityPickTerrainEffect;
@class UtilityModelEffect;
@class UtilityModelManager;


////////////////////////////////////////////////////////////////
// The only vertex attribute needed for terrain rendering is
// "position".
typedef enum
{
   TETerrainPositionAttrib,
   TETerrainNumberOfAttributes
} TETerrainAttribute;


@interface TETerrain (viewAdditions)

- (NSArray *)tiles;

- (void)prepareTerrainAttributes;

- (void)drawTerrainWithinTiles:(NSArray *)tiles
   withCamera:(UtilityCamera *)aCamera
   terrainEffect:(UtilityTerrainEffect *)aTerrainEffect;

- (void)drawModelsWithinTiles:(NSArray *)tiles
   withCamera:(UtilityCamera *)aCamera
   modelEffect:(UtilityModelEffect *)aModelEffect
   modelManager:(UtilityModelManager *)modelManager;

- (void)prepareToPickTerrain:(NSArray *)tiles
   withCamera:(UtilityCamera *)aCamera
   pickEffect:(UtilityPickTerrainEffect *)aPickEffect;

@end
```

```
//
//  TETerrain+modelAdditions.h
//
//

#import "TETerrain.h"
#import <GLKit/GLKit.h>


@interface TETerrain (modelAdditions)

- (GLfloat)calculatedHeightAtXPosMeters:(GLfloat)x
   zPosMeters:(GLfloat)z
   surfaceNormal:(GLKVector3 *)aNormal;
- (GLfloat)calculatedHeightAtXPos:(GLfloat)x
   zPos:(GLfloat)z
   surfaceNormal:(GLKVector3 *)aNormal;
- (GLfloat)heightAtXPos:(NSInteger)x zPos:(NSInteger)z;
- (GLfloat)heightAtXPosMeters:(GLfloat)x zPosMeters:(GLfloat)z;
- (GLfloat)maxHeightNearXPosMeters:(NSInteger)x
   zPosMeters:(NSInteger)z;
- (GLfloat)regionalHeightAtXPosMeters:(NSInteger)x
   zPosMeters:(NSInteger)z;
- (BOOL)isHeightValidAtXPos:(NSInteger)x zPos:(NSInteger)z;
- (BOOL)isHeightValidAtXPosMeters:(NSInteger)x
   zPosMeters:(NSInteger)z;

- (GLfloat)widthMeters;
- (GLfloat)heightMeters;
- (GLfloat)lengthMeters;

@end
```

类型 TETerrainAttribute 定义了用于渲染地形网格的 OpenGL ES 顶点属性。“-tiles”方法会返回一个瓦片数组，里面的每个瓦片引用了总地形的一个子集。这些瓦片是在“-tiles”方法第一次被调用时按需创建的。“-prepareTerrainAttributes”方法会配置用来渲染地形的 OpenGL ES 状态，包括顶点属性缓存的创建。在调用了“-prepareTerrainAttributes”方法后，接着调用“-drawTiles:(NSArray *)tiles”方法来渲染在 tiles 数组中的指定 tiles 所引用的地形网格部分。在类别中剩下的方法会检索这个地形的尺寸以及地形中各个位置的高度。

例子 OpenGLES_Ch10_1 通过 TETerrainTile 类封装了瓦片。TETerrainTile 的每个实例都会管理一个 OpenGL ES 元素数组缓存，这个缓存包含了整个地形的顶点子集的索引。元素数组缓存是在第 6 章讲解的。比较有意义的工作是在 TETerrainTile 类的“-draw”方法中执行的，这些工作包括绑定一个元素数组缓存并调用 glDrawElements()

函数以提交需要处理的顶点。

```
//
//  TETerrainTile.h
//
//

#import <Foundation/Foundation.h>

@class TETerrain;


@interface TETerrainTile : NSObject

@property (assign, nonatomic, readonly) NSInteger originX;
@property (assign, nonatomic, readonly) NSInteger originY;
@property (strong, nonatomic, readonly) NSSet *
   containedModelPlacements;

- (id)initWithTerrain:(TETerrain *)aTerrain
`  tileOriginX:(NSInteger)x
   tileOriginY:(NSInteger)y
   tileWidth:(NSInteger)aWidth
   tileLength:(NSInteger)aLength;

- (void)draw;

- (void)manageContainedModelPlacements:(NSSet *)somePlacements;
- (NSSet *)containedModelPlacements;

@end

static const NSInteger TETerrainTileDefaultWidth = 32;
static const NSInteger TETerrainTileDefaultLength = 32;
```

浏览在例子 OpenGLES_Ch10_1 中的 TETerrain+viewAdditions 类别和 TETerrainTile 类的实现。每个方法的实现都很简短。大部分代码与前面章节中管理和渲染网格的代码相似或者相同。

10.1.3 地形效果

这里的自定义 OpenGL ES Shading Language 程序是通过结合四个“细节”纹理来渲染地形的，“细节”纹理是根据存储在被称为 lightAndWeightsTextureInfo 的第五个纹理中的混合权重来混合的。当渲染网格时，可以把这些纹理联合起来产生类似隆起、涟漪、树叶、泥土，或者沙子的效果。lightAndWeightsTextureInfo 纹理还包含一个定向光的预计算烘焙灯光效果。第 4 章简短地介绍了把灯光烘焙进纹理的方法。下面的

UtilityTerrainEffect 类会管理用于地形渲染的 OpenGL ES Shading Language 程序。

```
//
//  UtilityTerrainEffect.h
//
//

#import "UtilityEffect.h"
#import <GLKit/GLKit.h>

@class TETerrain;
@class UtilityTextureInfo;


@interface UtilityTerrainEffect : UtilityEffect

@property (assign, nonatomic, readwrite)
   GLKVector4 globalAmbientLightColor;
@property (assign, nonatomic, readwrite)
   GLKMatrix4 projectionMatrix;
@property (assign, nonatomic, readwrite)
   GLKMatrix4 modelviewMatrix;
@property (assign, nonatomic, readwrite)
   GLKMatrix3 textureMatrix0;
@property (assign, nonatomic, readwrite)
   GLKMatrix3 textureMatrix1;
@property (assign, nonatomic, readwrite)
   GLKMatrix3 textureMatrix2;
@property (assign, nonatomic, readwrite)
   GLKMatrix3 textureMatrix3;
@property (strong, nonatomic, readwrite)
   UtilityTextureInfo *lightAndWeightsTextureInfo;
@property (strong, nonatomic, readwrite)
   UtilityTextureInfo *detailTextureInfo0;
@property (strong, nonatomic, readwrite)
   UtilityTextureInfo *detailTextureInfo1;
@property (strong, nonatomic, readwrite)
   UtilityTextureInfo *detailTextureInfo2;
@property (strong, nonatomic, readwrite)
   UtilityTextureInfo *detailTextureInfo3;

// Designated initializer
- (id)initWithTerrain:(TETerrain *)aTerrain;

- (void)prepareToDraw;

@end
```

在 UtilityTerrainEffect 类中 Shading Language 程序会使用被称为 textureMatrix0、

textureMatrix1、textureMatrix2 和 textureMatrix3 的纹理矩阵来分别变换每个纹理，纹理矩阵 textureMatrix0、textureMatrix1、textureMatrix2 和 textureMatrix3 会分别影响 detailTextureInfo0 到 detailTextureInfo3。在使用 TETerrain 的“-drawTiles:”方法提交地形顶点属性之前，会调用 UtilityTerrainEffect 的“-prepareToDraw”方法。

注意 例子 OpenGLES_Ch10_1 实现了多个“Effect”类，这些类封装了分别用于地形渲染、模型渲染和拾取的 Shading Language 程序。为了避免不必要的代码复制，例子 OpenGLES_Ch10_1 会使用抽象基类 UtilityEffect 来加载、编译、链接和验证 Shading Language 程序。UtilityEffect 遵守 GLKit 的 GLKNamedEffect 协议。在 Objective-C 语言环境中，UtilityEffect 是“抽象的”，因为 UtilityEffect 实现了“-bindAttribLocations”和“ –configureUniformLocations”方法，以便当它们被直接调用时可以生成异常。UtilityEffect 的子类必须要重写这两个方法。

下面用于地形渲染的 Shading Language 程序会使用与前面的顶点着色器例子相似的一个 model-view 投影结合矩阵来变换顶点位置属性。在这里，顶点位置属性还被用作纹理坐标，以使细节纹理自动覆盖整个地形网格。但是，着色器会使用单独的矩阵来变换每个细节纹理的纹理坐标。每个纹理可能会被单独的施加平移、旋转和缩放变换。细节纹理的缩放会控制这个纹理在整个网格中重复次数。例如，纹理可能会被缩放以在地形上重复数百次，也可能精确地匹配地形而没有重复。纹理变换和重复是在第 3 章讲解的。回顾第 3 章的例子有助于理解在本例中的较复杂变换。

```
//
//  UtilityTerrainShader.vsh
//
//

////////////////////////////////////////////////////////////////////////
// VERTEX ATTRIBUTES
////////////////////////////////////////////////////////////////////////
attribute vec3 a_position;

////////////////////////////////////////////////////////////////////////
// TEXTURE
////////////////////////////////////////////////////////////////////////
#define MAX_TEXTURES    5
#define MAX_TEX_COORDS  5

////////////////////////////////////////////////////////////////////////
// UNIFORMS
////////////////////////////////////////////////////////////////////////
uniform highp mat4      u_mvpMatrix;
uniform highp mat3      u_texMatrices[MAX_TEXTURES];
```

```
uniform sampler2D       u_units[MAX_TEXTURES];
uniform lowp vec4       u_globalAmbientColor;

//////////////////////////////////////////////////////////////////
// Varyings
//////////////////////////////////////////////////////////////////
varying highp vec2      v_texCoords[MAX_TEX_COORDS];


void main()
{
   vec3 coords = u_texMatrices[0] * a_position;
   v_texCoords[0] = vec2(coords.x, coords.z);
   coords = u_texMatrices[1] * a_position;
   v_texCoords[1] = vec2(coords.x, coords.z);
   coords = u_texMatrices[2] * a_position;
   v_texCoords[2] = vec2(coords.x, coords.z);
   coords = u_texMatrices[3] * a_position;
   v_texCoords[3] = vec2(coords.x, coords.z);
   coords = u_texMatrices[4] * a_position;
   v_texCoords[4] = vec2(coords.x, coords.z);

   gl_Position = u_mvpMatrix * vec4(a_position, 1.0);
}
```

使用权重因数混合纹理颜色与在第 7 章中介绍的使用权重来影响顶点位置的方式大致相同。UtilityTerrainEffect 和 UtilityTerrainShader 会使用 detailTextureInfo3 来提供默认的地形片元颜色。换句话说，如果所有的纹理混合权重都是零，那么地形的每个片元颜色都 100% 来自 detailTextureInfo3 内的纹素。基于权重混合的工作方式就像是在第 3 章讲解的 GL_ONE_MINUS_SRC_ALPHA 混合函数。OpenGL ES Shading Language 甚至提供了一个内置的叫做 mix(first_color, second_color, weight) 的函数来返回表达式 ((1.0 - weight) * first_color) + (weight * second_color) 的值。在下面的片元着色器中，由 detailTextureInfo3 提供的默认 textureColor 会基于 weights.x 来与 textureColor0 混合，weights.x 对应于 lightAndWeightsTextureInfo 纹素的红色分量。接着，使用 weights.y 把 textureColor1 混合进来，weights.y 对应于 lightAndWeightsTextureInfo 纹素的绿色分量。最后，使用 weights.z 把 textureColor2 混合进来，weights.z 对应于 lightAndWeightsTextureInfo 纹素的蓝色分量。

在 lightAndWeightsTextureInfo 中的纹素的最后一个颜色分量通常被解析为透明度或者在一个（X，Y，Z，W）坐标系中的 W，不过在这里，最后一个分量定义了每个渲染片元的灯光强度。灯光强度是在 Mac OS X 平台上的 TerrainEditor 应用中预计算的，存储在由例子 OpenGLES_Ch10_1 所加载的 lightAndWeightsTextureInfo 中。

```
//
//  UtilityTerrainShader.fsh
//
//

//////////////////////////////////////////////////////////////////////
// TEXTURE
//////////////////////////////////////////////////////////////////////
#define MAX_TEXTURES    5
#define MAX_TEX_COORDS  5

//////////////////////////////////////////////////////////////////////
// UNIFORMS
//////////////////////////////////////////////////////////////////////
uniform highp mat4      u_mvpMatrix;
uniform highp mat3      u_texMatrices[MAX_TEXTURES];
uniform sampler2D       u_units[MAX_TEXTURES];
uniform lowp vec4       u_globalAmbientColor;

//////////////////////////////////////////////////////////////////////
// Varyings
//////////////////////////////////////////////////////////////////////
varying highp vec2      v_texCoords[MAX_TEX_COORDS];

void main()
{
   // Extract light color from w component of light and weight
   // texture
   lowp vec4 lightAndWeights =
      texture2D(u_units[0], v_texCoords[0]);
   lowp vec4 lightColor = u_globalAmbientColor + lowp vec4(
      lightAndWeights.w,
      lightAndWeights.w,
      lightAndWeights.w,
      1.0);

   // Extract texture mixing weights from light and weight
   // texture
  lowp vec3 weights = lowp vec3(
     lightAndWeights.x,
     lightAndWeights.y,
     lightAndWeights.z);

  // Blend the terrain textures using weights
  lowp vec4 textureColor0 = texture2D(u_units[1], v_texCoords[1]);
  lowp vec4 textureColor1 = texture2D(u_units[2], v_texCoords[2]);
  lowp vec4 textureColor2 = texture2D(u_units[3], v_texCoords[3]);
  lowp vec4 textureColor3 = texture2D(u_units[4], v_texCoords[4]);
```

```
    lowp vec4 textureColor = textureColor3;
    textureColor = mix(textureColor, textureColor0, weights.x);
    textureColor = mix(textureColor, textureColor1, weights.y);
    textureColor = mix(textureColor, textureColor2, weights.z);

    // Scale by light color
    lowp vec4 color = lightColor * textureColor;

    // Final terrain color is always opaque
    color.a = 1.0;

    gl_FragColor = color;
}
```

截止到现在这个例子的 TETerrain、TETerrainTile 和 UtilityTerrainEffect 类会共同努力来有效地渲染复杂的地形网格。10.5 节会讨论怎么使用在第 9 章介绍的剔除和场景图技术来改善地形渲染性能的问题。

10.2　添加模型

图 10-5 是在一个场景中一起渲染的地形和模型的屏幕截图。例子 OpenGLES_Ch10_1 会使用与前面例子中的类相似的 UtlityMesh 和 UtilityModel 类。例子 OpenGLES_Ch10_1 向 UtilityMesh 类添加了一个“-prepareToPick”方法。这个新方法的工作方式类似于现存的“-prepareToDraw”方法，只不过没有启用类似法向量和纹理坐标的顶点属性，因为在拾取中不需要这些属性。

doesRequireLighting 属性被添加到了 UtilityModel 类中。一些模型，包括树木，会具有烘焙进它们纹理的灯光效果。doesRequireLighting 属性用于区分使用 OpenGL 灯光模型渲染的模型和使用烘焙灯光渲染的模型。

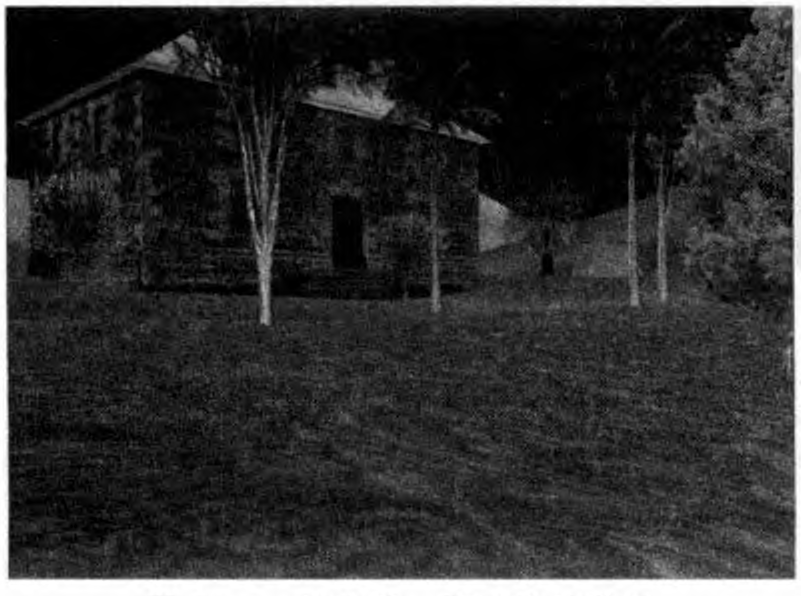


图 10-5　与地形一起渲染的模型

10.2.1 模型放置

TEModelPlacement 类是 Xcode 使用在图 10-4 中显示的数据定义所生成的，存储了模型的名字、位置和围绕 Y 轴的旋转角度属性。与地形一起存储的 TEModelPlacement 实例的集合指定了树木、石头和建筑物的位置。例如，在图 10-5 中的相同的棕榈树模型会渲染多次，因为多个 TEModelPlacement 实例在地形内的不同位置引用了棕榈树模型的名字。

10.2.2 模型效果

例子 OpenGLES_Ch10_1 的地形一直具有烘焙灯光和多重纹理。与此相反，模型会使用一个单独的纹理，并且可能会根据情况带有或者不带有烘焙灯光。UtilityModelEffect 类会管理做了优化的用于渲染带有或者不带有烘焙灯光的模型的 OpenGL ES Shading Language 程序。

```
//
//  UtilityModelEffect.h
//
//

#import "UtilityEffect.h"
#import <GLKit/GLKit.h>


@interface UtilityModelEffect : UtilityEffect

@property (assign, nonatomic, readwrite) GLKMatrix4
   projectionMatrix;
@property (assign, nonatomic, readwrite) GLKMatrix4
   modelviewMatrix;
@property (assign, nonatomic, readwrite) GLKVector4
   globalAmbientLightColor;
@property (assign, nonatomic, readwrite) GLKVector3
   diffuseLightDirection;
@property (assign, nonatomic, readwrite) GLKVector4
   diffuseLightColor;
@property (strong, nonatomic, readwrite) GLKTextureInfo *
   texture2D;

- (void)prepareLightColors;
- (void)prepareModelview;
- (void)prepareModelviewWithoutNormal;

@end
```

当渲染带有灯光的模型时，UtilityModelEffect 的 diffuseLightColor 和 globalAmbientLightColor 属性被设置为与 TETerrainEffect 的值相匹配。UtilityModelEffect 的 diffuseLightDirection 属性用于设置灯光方向，具体会与烘焙灯光到地形的 lightAndWeightsTextureInfo 中所使用的灯光方向相匹配。灯光方向是由 TETerrain 对象存储的。当渲染不带灯光的模型时，diffuseLightColor 被设置为黑色，globalAmbientLightColor 被设置为全白，这样做可以避免 OpenGL ES 灯光方程式更改这个模型的纹理颜色。

用于渲染模型的自定义 Shading Language 程序——UtilityModelShader.vsh 和 UtilityModelShader.fsh，仍然比较简单。UtilityModelShader.vsh 的相关部分会向片元程序提供纹理坐标、变换的位置坐标，以及为一个单独的定向光计算的灯光效果。

```
void main()
{
   // Texture coords
   v_texCoords[0] = a_texCoords0;

   // Lighting
   lowp vec3 normal = normalize(u_normalMatrix * a_normal);
   lowp float nDotL = max(
      dot(normal, normalize(u_diffuseLightDirection)), 0.0);
   v_lightColor = (nDotL * u_diffuseLightColor) +
      u_globalAmbient;

   gl_Position = u_mvpMatrix * vec4(a_position, 1.0);
}
```

下面的 UtilityModelShader.fsh 程序会使用一个技巧来正确地渲染带有半透明纹理的模型。第 8 章讲解过渲染半透明纹理会产生难看的可视伪影，除非这些有纹理的对象是按照其与平截体近面的距离大小顺序来渲染的。但是，当纹理只包含完全透明和完全不透明的纹素（没有半透明）时，可以有不同排序。

```
void main()
{
   lowp vec4 color = texture2D(u_units[0], v_texCoords[0]);
   if (color.a < 0.2)
   {  // discard nearly transparent fragments
      discard;
   }

   gl_FragColor = color * v_lightColor;
}
```

在 UtilityModelShader.fsh 中的粗体代码会测试当前片元的颜色是否为几乎透明的，如果是的话就丢弃这些片元。一个丢弃的片元既不会影响像素颜色渲染缓存，又不会影响深度渲染缓存。这正是渲染类似在图 10-5 中显示的树木模型时所期望的结果。丢弃

的透明片元不会遮挡其他 3D 对象产生的片元。这个技巧并不适合半透明纹素，因为在像素颜色渲染缓存中的最终像素颜色取决于多重 3D 对象生成的片元的混合。在这种情况下，片元的生成顺序变得至关重要，并且排序也是必要的。

注意 “if”语句的条件执行和 discard 命令的使用是计算最密集的，因此它们会减慢 GPU 执行运算的速度。例子 OpenGLES_Ch10_1 的实现是时间、复杂度与计算密集度之间的一个折中，具体是指基于与观察者的距离来排序模型所需的时间和复杂度与 Shading Language 程序的计算密集度之间的折中。不同的应用可能会有不同的折中基准。

10.3 OpenGL ES 摄像机

视平截体定义了在 5.6 节讲解过的，并在第 9 章详细说明的“视点”。很多 3D 应用会使用一个比喻的“摄像机”，通过这个“摄像机”来查看场景。使用一个虚拟的摄像机在虚拟空间中移动，而不是直接使用一个平截体数据结构。当需要的时候，与摄像机的位置、方向和视野对应的平截体可以计算出来。

摄像机是用于定义视点的一个非常有用的比喻，使用这个比喻便于创建类似下面的在 OpenGLES_Ch10_1 中的 UtilityCamera 类的类。UtilityCamera 类有类似“-(void) setPosition:(GLKVector3)aPosition lookAtPosition:(GLKVector3)lookAtPosition” 和“-(void) moveBy:(GLKVector3) aVector”的方法。可以使用比直接处理平截体数据结构更熟悉和直观的动作和术语在虚拟世界中移动摄像机，以此来活动视点。

```
//
//  UtilityCamera.h
//
//

#include <GLKit/GLKit.h>
#import "AGLKFrustum.h"

@class UtilityCamera;

@protocol UtilityOpenGLCameraDelegate <NSObject>

////////////////////////////////////////////////////////////////
// Returning NO prevents changes.
@optional
- (BOOL)camera:(UtilityCamera *)aCamera
   willChangeEyePosition:(GLKVector3 *)eyePositionPtr
   lookAtPosition:(GLKVector3 *)lookAtPositionPtr;
```

```
@end


@interface UtilityCamera : NSObject

@property (nonatomic, assign, readwrite)
   __unsafe_unretained IBOutlet id delegate;
@property(assign, nOnatomic, readonly)
   GLKMatrix4 projectionMatrix;
@property(assign, nonatomic, readonly)
   GLKMatrix4 modelviewMatrix;
@property(assign, nonatomic, readonly)
   GLKVector3 position;
@property(assign, nonatomic, readonly)
   GLKVector3 lookAtPosition;
@property(assign, nonatomic, readonly)
   GLKVector3 upUnitVector;
@property(nonatomic, readonly)
   const AGLKFrustum *frustumForCulling;

- (void)configurePerspectiveFieldOfViewRad:(GLfloat)angle
   aspectRatio:(GLfloat)anAspectRatio
   near:(GLfloat)nearLimit
   far:(GLfloat)farLimit;

- (void)rotateAngleRadiansAboutY:(GLfloat)anAngleRadians;
- (void)rotateAngleRadiansAboutX:(GLfloat)anAngleRadians;

- (void)moveBy:(GLKVector3)aVector;

- (void)setPosition:(GLKVector3)aPosition
    lookAtPosition:(GLKVector3)lookAtPosition;
- (void)setOrientation:(GLKMatrix4)aMatrix;

@end
```

UtilityCamera 类重用并封装了在第 9 章介绍的 AGLKFrustum 数据结构。Utility-Camera 的 projectionMatrix 和 modelviewMatrix 属性是直接使用与摄像机的位置、方向和视野相对应的平截体计算出来的。

所有的用于改变平截体的位置和方向的 UtilityCamera 方法都会调用 UtilityCamera 的“-setPosition:lookAtPosition:”方法，然后这个方法转而向摄像机的委托发送“- (BOOL) camera:(UtilityCamera *)aCamera willChangeEyePosition:(GLKVector3 *)eyePositionPtr lookAtPosition:(GLKVector3 *)lookAtPositionPtr”消息。这个委托会执行额外的处理以控制要发生什么。这个委托可以返回 NO 来阻止摄像机位置的变化。

注意 委托是 Cocoa Touch 的一般模式。委托是提供了影响另一个对象行为的机会的对象。基本概念是两个对象协调解决一个问题。其中一个对象是非常综合的，并且适合在各种情况下重用。它会存储对于另一个对象的引用，即它的委托，并会在关键时刻向它的委托发送消息。这个消息给予了这个委托一个执行额外处理或者控制要发生什么的机会。应该在委托中实现特定于应用的规则和逻辑，而不是修改或者子类化更通用的对象。

例子 OpenGLES_Ch10_1 会使用这个应用的 OpenGLES_Ch10_1ViewController 实例作为摄像机的委托。OpenGLES_Ch10_1ViewController 实现了“-camera:willChangeEyePosition:lookAtPosition:”方法来防止摄像机的位置低于在摄像机位置处的地形高度。这个委托的更精密实现还能防止摄像机与其他模型碰撞，或者实现虚拟的重力以让摄像机滑下地形内的丘陵。

下面的 UtilityCamera 类的部分实现代码解释了这个类是怎么管理视平截体以及怎么与一个委托对象通信的。要获得完整实现，请访问：http://opengles.cosmicthump.com/learning-opengl-es-sample-code/。

```
//
//  UtilityCamera.m
//
//

#import "UtilityCamera.h"
#import "AGLKFrustum.h"


@interface UtilityCamera ()
{
   AGLKFrustum frustum;
}

@property(assign, nonatomic, readwrite)
   BOOL isInCallback;

@end


////////////////////////////////////////////////////////////////////////
// Instances of this class encapsulate a viewing frustum and
// enable point-of-view animation using the metaphor of a
// camera.
@implementation UtilityCamera

@synthesize delegate = delegate_;
```

```
@synthesize isInCallback = isInCallback_;


////////////////////////////////////////////////////////////////////////
// Initialize the receiver with a default frustum
- (id)init
{
   self = [super init];
   if(nil != self)
   {
      // Default 45 deg. field of view, square aspect ratio,
      // near distance of 0.5, and far distance of 5000
      frustum = AGLKFrustumMakeFrustumWithParameters(
         GLKMathDegreesToRadians(45.0f),
         1.0f,
         0.5f,
         5000.0f);
      // Default eye at origin, look down neg. Z axis,
      // Y axis is "up"
      AGLKFrustumSetPositionAndDirection(
         &frustum,
         GLKVector3Make(0.0f, 0.0f, 0.0f),
         GLKVector3Make(0.0f, 0.0f, -1.0f),
         GLKVector3Make(0.0f, 1.0f, 0.0f));
   }

   return self;
}


////////////////////////////////////////////////////////////////////////
// Set the receiver's frustum shape
- (void)configurePerspectiveFieldOfViewRad:(GLfloat)angle
   aspectRatio:(GLfloat)anAspectRatio
   near:(GLfloat)nearLimit
   far:(GLfloat)farLimit;
{
   AGLKFrustumSetPerspective(
      &frustum,
      angle,
      anAspectRatio,
      nearLimit,
      farLimit);
}


////////////////////////////////////////////////////////////////////////
// Set the receiver's eye and look-at positions and give optional
```

```
// delegate object opportunity to revise or constrain the
// specified positions.
- (void)setPosition:(GLKVector3)aPosition
   lookAtPosition:(GLKVector3)lookAtPosition;
{
   if(!self.isInCallback)
   {  // Prevent recursive call to -setPosition:lookAtPosition:
      self.isInCallback = YES;
      BOOL shouldMakeChange = YES;

      if([self.delegate respondsToSelector:
         @selector(camera:willChangeEyePosition:lookAtPosition:)])
      {
         shouldMakeChange =
            [self.delegate camera:self
               willChangeEyePosition:&aPosition
               lookAtPosition:&lookAtPosition];
      }

      if(shouldMakeChange)
      {
         const GLKVector3 upUnitVector =
            GLKVector3Make(0.0f, 1.0f, 0.0f); // Assume Y up

         AGLKFrustumSetPositionAndDirection(
               &frustum,
               aPosition,
               lookAtPosition,
               upUnitVector);
      }

      self.isInCallback = NO;
   }
}


////////////////////////////////////////////////////////////////
// Return's the receiver's look-at position
- (GLKVector3)lookAtPosition
{
   const GLKVector3 eyePosition = frustum.eyePosition;
   const GLKVector3 lookAtPosition = GLKVector3Add(
      eyePosition, frustum.zNormalVector);

   return lookAtPosition;
}
```

```
//////////////////////////////////////////////////////////////////
// Move the receiver's eye position and look-at position by the
// direction and distance of aVector. This method calls
// -setPosition:lookAtPosition: giving the receiver's optional
// delegate an opportunity to revise or constrain the resulting
// frustum positions.
- (void)moveBy:(GLKVector3)aVector;
{
   const GLKVector3 currentEyePosition = [self position];
   const GLKVector3 currentLookAtPosition = [self lookAtPosition];

   [self setPosition:GLKVector3Add(currentEyePosition, aVector)
      lookAtPosition:GLKVector3Add(currentLookAtPosition, aVector)];
}

@end
```

10.4　拾取

例子 OpenGLES_Ch10_1 使用专门的伪色彩渲染来实现拾取。当识别到一个用户触摸时，会使用 UtilityPickTerrainEffect 类来把场景渲染到一个像素颜色渲染缓存。地形的 X 和 Z 位置坐标被编码到片元的红和绿颜色分量中。例如，基于地形网格的 {150，45，78} 位置生成的片元的颜色为浮点 RGB 颜色 {150.0 / terrainWidth, 78.0 / terrainLength, 0.0}。然后从像素颜色渲染缓存中读取触摸位置之下的像素颜色。检测这个像素颜色的红和绿分量以确定触摸到的地形的位置。

用户是看不到伪色彩渲染的，因为用于拾取的像素颜色渲染缓存永远不会显示到屏幕上。无论摄像机的位置和方向怎么变化，这个方法都有效。在前景中渲染的地形会自动地挡住背景中的地形，因此返回的总是触摸位置处的与视点最近的地形位置。图 10-6 用于帮助形象化伪色彩场景。

图 10-6　把编码了位置数据中的片元颜色渲染出来

UtilityPickTerrainEffect 会渲染到一个 OpenGL ES 帧缓存对象（frame buffer object-FBO）中。帧缓存是在第 1 章介绍的。帧缓存会接收渲染结果。截止到现在的例子使用的都是用于显示到屏幕上的帧缓存。Cocoa Touch 的 GLKView 类创建了一个默认的帧缓存和一个与视图的 Cocoa Touch CALayer 共享的附加像素颜色渲染缓存。UtilityPickTerrainEffect 类

会创建一个附加的帧缓存，这个帧缓存是 FBO 类型的，而不是 CALayers。

下面的 UtilityPickTerrainInfo 数据类型和 UtilityPickTerrainEffect 类接口展示了拾取有多么简单。在“-initWithTerrain:”方法中使用地形初始化了一个 UtilityPickTerrainEffect 实例后，调用这个实例用于确定一个触摸事件位置的“-terrainInfoForProjectionPosition:”方法，这个调用会返回包含这个位置所对应的地形 X 和 Z 坐标的一个 UtilityPickTerrainInfo 结构。

```
//
//  UtilityPickTerrainEffect.h
//
//
#import "UtilityEffect.h"
#import <GLKit/GLKit.h>

@class TETerrain;
@class TEModelManager;


typedef struct
{
   GLKVector2 position;
   unsigned char modelIndex;
}
UtilityPickTerrainInfo;


@interface UtilityPickTerrainEffect : UtilityEffect

@property(assign, nonatomic, readwrite)
   GLKMatrix4 projectionMatrix;
@property(assign, nonatomic, readwrite)
   GLKMatrix4 modelviewMatrix;
@property(assign, nonatomic, readwrite)
   unsigned char modelIndex;

// Designated initializer
- (id)initWithTerrain:(TETerrain *)aTerrain;

- (UtilityPickTerrainInfo)terrainInfoForProjectionPosition:
   (GLKVector2)aPosition;

@end
```

传递给“-terrainInfoForProjectionPosition:”的位置是处在相对于被触摸的 GLKView 的位置和大小的 Cocoa Touch 坐标系中的。返回的 UtilityPickTerrainInfo 提供了在 3D 虚拟世界中的对应坐标。

UtilityPickTerrainEffect 的 FBO 没有必要与用于在屏幕上显示的帧缓存的大小相匹配。实际上，一个 FBO 的最佳大小是让尺寸为 2 的幂。无论用于在屏幕上显示的帧缓存的大小是多少，UtilityPickTerrainEffect 的 FBO 的大小都是 512×512 的。2 的幂的尺寸是在第 3 章讲解的，这是非常重要的，因为每个 FBO 通常都会与一个 OpenGL ES 纹理缓存共享像素颜色渲染缓存存储空间。渲染到一个 FBO 通常叫做“渲染到纹理”，并且产生的纹理可以用于接下来的渲染。例子 OpenGLES_Ch10_1 没有重用与 FBO 相关联的纹理缓存，不过苹果提供了 GLEssentials 示例程序，这个例子会使用一个 FBO 来渲染在一个场景自身内的场景倒影。这个例子的下载地址为：http://developer.apple.com/library/ios/samplecode/GLEssentials/Introduction/Intro.html。

UtilityPickTerrainEffect 类使用下面的方法创建并配置了它的用于拾取的 FBO。这段代码与例子 GLEssentials 和用来介绍 GLKView 的实现方式的例子 OpenGLES_Ch2_2 中的代码相似。用来创建作为 FBO 的像素颜色渲染缓存的纹理的代码与用于介绍 GLKTextureLoader 的工作方式的例子 OpenGLES_Ch3_2 中的代码相似。

```
////////////////////////////////////////////////////////////////
// Build a Frame Buffer Object with attached Pixel Color Render
// Buffer and Depth Buffer to receive the results of rendering
// in false color for picking.
-(GLuint)buildFBOWithWidth:(GLuint)fboWidth
                    height:(GLuint)fboHeight
{
   GLuint fboName;
   GLuint colorTexture;

   // Create a texture object to apply to model
   glGenTextures(1, &colorTexture);
   glBindTexture(GL_TEXTURE_2D, colorTexture);

   // Set up filter and wrap modes for this texture object
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                   GL_CLAMP_TO_EDGE);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                   GL_CLAMP_TO_EDGE);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                   GL_LINEAR);
   glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                   GL_LINEAR_MIPMAP_LINEAR);

   // Allocate a texture image we can render into
   // Pass NULL for the data parameter since we don't need to
   // load image data. We will be generating the image by
   // rendering to this texture.
   glTexImage2D(GL_TEXTURE_2D,
                0,
```

```
                GL_RGBA,
                fboWidth,
                fboHeight,
                0,
                GL_RGBA,
                GL_UNSIGNED_BYTE,
                NULL);

    GLuint depthRenderbuffer;
    glGenRenderbuffers(1, &depthRenderbuffer);
    glBindRenderbuffer(GL_RENDERBUFFER, depthRenderbuffer);
    glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT16,
                          fboWidth, fboHeight);

    glGenFramebuffers(1, &fboName);
    glBindFramebuffer(GL_FRAMEBUFFER, fboName);
    glFramebufferTexture2D(GL_FRAMEBUFFER,
                           GL_COLOR_ATTACHMENT0,
                           GL_TEXTURE_2D, colorTexture, 0);
    glFramebufferRenderbuffer(GL_FRAMEBUFFER,
       GL_DEPTH_ATTACHMENT, GL_RENDERBUFFER, depthRenderbuffer);

    if(glCheckFramebufferStatus(GL_FRAMEBUFFER) !=
       GL_FRAMEBUFFER_COMPLETE)
    {
       NSLog(@"failed to make complete framebuffer object %x",
          glCheckFramebufferStatus(GL_FRAMEBUFFER));

       UtilityPickTerrainEffectDestroyFBO(fboName);

       fboName = 0;
    }

    return fboName;
}
```

UtilityPickTerrainEffect 会使用下面的两个方法销毁它的 FBO 并把资源返回给 OpenGL ES，这两个方法与苹果公司提供的例子 GLEssentials 中的代码是相似的。

```
////////////////////////////////////////////////////////////////
// This function deletes the specified FBO including all of its
// attachments and returns resources to OpenGL
static void UtilityPickTerrainEffectDestroyFBO(GLuint fboName)
{
   if(0 == fboName)
   {
      return;
   }

   glBindFramebuffer(GL_FRAMEBUFFER, fboName);
```

```
   // Delete the attachment
   UtilityPickTerrainEffectDeleteFBOAttachment(
      GL_COLOR_ATTACHMENT0);

   // Delete any depth or stencil buffer attached
   UtilityPickTerrainEffectDeleteFBOAttachment(
      GL_DEPTH_ATTACHMENT);
   glDeleteFramebuffers(1, &fboName);
}

////////////////////////////////////////////////////////////////////////
// This function deletes the specified attachment and returns
// resources to OpenGL
static void UtilityPickTerrainEffectDeleteFBOAttachment(
   GLenum attachment)
{
   GLint param;
   GLuint objName;

   glGetFramebufferAttachmentParameteriv(
      GL_FRAMEBUFFER,
      attachment,
      GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE,
      &param);

   if(GL_RENDERBUFFER == param)
   {
      glGetFramebufferAttachmentParameteriv(
         GL_FRAMEBUFFER,
         attachment,
         GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME,
         &param);

      objName = ((GLuint*)(&param))[0];
      glDeleteRenderbuffers(1, &objName);
   }
   else if(GL_TEXTURE == param)
   {
      glGetFramebufferAttachmentParameteriv(
         GL_FRAMEBUFFER,
         attachment,
         GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME,
         &param);

      objName = ((GLuint*)(&param))[0];
      glDeleteTextures(1, &objName);
   }
}
```

UtilityPickTerrainEffect 的“-prepareToDraw”方法会设置让 OpenGL ES 渲染到 FBO 中，参见下面的代码。

```
//////////////////////////////////////////////////////////////////
// Prepare OpenGL state for rendering into the FBO
- (void)prepareToDraw;
{
   [super prepareToDraw];
   glBindFramebuffer(GL_FRAMEBUFFER, pickFBO);
   glViewport(0, 0, UtilityPickTerrainFBOWidth,
      UtilityPickTerrainFBOHeight);
}
```

在使用 UtilityPickTerrainEffect 渲染了这个场景后，使用“-terrainInfoForProjection Position:”方法来读取在 0.0 到 1.0 范围内的指定位置的像素颜色，然后把这个像素颜色还原为 3D 的 X 和 Z 坐标：

```
//////////////////////////////////////////////////////////////////
// This method returns the 3D X,Z coordinates of any terrain
// at aPosition. The aPosition coordinates must be in the range
// 0.0 to 1.0 corresponding to the relative location of aPosition
// within a Cocoa Touch view a.k.a. "projection" coordinates.
- (UtilityPickTerrainInfo)terrainInfoForProjectionPosition:
   (GLKVector2)aPosition
{
   GLubyte pixelColor[4];  // Red, Green, Blue, Alpha color
   GLint readLocationX = MIN((UtilityPickTerrainFBOWidth - 1),
      (UtilityPickTerrainFBOWidth - 1) * aPosition.x);
   GLint readLocationY = MIN((UtilityPickTerrainFBOHeight - 1),
      (UtilityPickTerrainFBOHeight - 1) * aPosition.y);
   glReadPixels(readLocationX,
      readLocationY,
      1,
      1,
      GL_RGBA,
      GL_UNSIGNED_BYTE,
      pixelColor);

   UtilityPickTerrainInfo result;
   GLKVector2 position = {
      self.width * (GLfloat)pixelColor[0] /  // red component
         UtilityPickTerrainMaxIndex,
      self.length * (GLfloat)pixelColor[1] / // green component
         UtilityPickTerrainMaxIndex
   };
   result.position = position;
   result.modelIndex = pixelColor[2]; // blue component

   return result;
}
```

注意　把渲染值从 FBO 读回 CPU 控制的内存的过程可能需要执行费时的同步操作。CPU 必须要等待 GPU 渲染完场景，而且在 GPU 能够接着渲染其他的东西到 FBO 之前，GPU 必须要等待 CPU 读取完成。OpenGL ES 会自动管理所有必要的同步，但是永远无法完全避免同步延迟。拾取在每秒中至多发生几次，而理想情况下普通渲染每秒可以发生 30 或者 60 次。例子 OpenGLES_Ch10_1 可以接受一个基于颜色拾取的折中方案，但是其他的应用可能需要使用不同的折中方案。

最后一点，就像 UtilityPickTerrainEffect 会把地形顶点坐标编码到每个渲染像素的红色和绿色分量中，可选的 modelIndex 是编码到蓝色分量中的。如果从 FBO 读取的这个像素的蓝色分量的值不是 0，那么这个蓝色分量会被缩放到 1 到 255 的范围内，并会对应于在拾取位置的模型的索引。根据编码方案，在任何一个场景中，任意的一个时刻至多能有 255 个可用于拾取的模型，但这已经足以满足例子 OpenGLES_Ch10_1 所需的模型的数量。

联合使用 UtilityPickTerrainEffect 和下面的 OpenGL ES 2.0 Shading Language 程序来使用用于拾取的伪色彩去渲染地形和模型：

```
//
//  UtilityPickTerrainShader.vsh
//
//

////////////////////////////////////////////////////////////////////////
// VERTEX ATTRIBUTES
////////////////////////////////////////////////////////////////////////
attribute vec3 a_position;

////////////////////////////////////////////////////////////////////////
// TEXTURE
////////////////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////////////////
// UNIFORMS
////////////////////////////////////////////////////////////////////////
uniform highp mat4      u_mvpMatrix;
uniform highp vec2      u_dimensionFactors;
uniform lowp float      u_modelIndex;

////////////////////////////////////////////////////////////////////////
// Varyings
////////////////////////////////////////////////////////////////////////
```

```
varying lowp vec4       v_color;


void main()
{
   float r = a_position.x * u_dimensionFactors.x;
   float g = a_position.z * u_dimensionFactors.y;

   v_color = vec4(r, g, u_modelIndex, 1.0);

   gl_Position = u_mvpMatrix * vec4(a_position, 1.0);
}

//
//  UtilityPickTerrainShader.fsh
//
//

////////////////////////////////////////////////////////////////
// UNIFORMS
////////////////////////////////////////////////////////////////
uniform highp mat4      u_mvpMatrix;
uniform highp vec2      u_dimensionFactors;
uniform lowp float      u_modelIndex;

////////////////////////////////////////////////////////////////
// Varyings
////////////////////////////////////////////////////////////////
varying lowp vec4       v_color;


void main()
{
   gl_FragColor = v_color;
}
```

备选拾取方式

从历史上说，OpenGL 的桌面版支持使用 glRenderMode(GL_SELECT) 函数来为拾取设定一个专门的渲染模式。现在不提倡使用 GL_SELECT 拾取模式，因为现在很少有 GPU 能够从硬件上支持这个功能。使用 GL_SELECT 渲染模式通常会迫使 OpenGL 退化到基于软件的渲染，即运行在 CPU 上而不是 GPU 上，同时在这个过程中 GPU 会处于闲置状态。

纯粹的几何拾取方式可以完全避免任何额外渲染。最常见的几何方式叫做光线投射法（ray casting）。设想一个光线从平截体近面上的一个触摸位置投射向与这个触摸位置对应的平截体远面上的一个点。被这个光线穿过的离视点最近的对象就是要拾取的对象。

不幸的是，完全使用类似光线投射法的几何方式通常会带来很多的软件复杂性并且相交对象的检索可能需要高昂的计算代价。除非用于检测相交的计算执行得非常快速，并且要测试的对象相对较少，要不然几何方式可能要比颜色拾取方式带来更多的延迟。

物理引擎 Bullet 3D Game Multiphysics 提供了最先进的光线和 3D 对象之间的碰撞检测方法。如果你已经在你的应用中使用过 Bullet 3D 来实现类似重力加速和对象碰撞检测的物理效果，那么你应该知道这个基于光线投射法的拾取几乎是没有性能损失的。Bullet Collision Detection 可以独立于 Bullet 3D 库来单独使用。很多的 iOS 应用会使用 Bullet 3D，它可以免费用于商业用途，并且遵循开源 ZLib 许可条款：http://code.google.com/p/bullet/。

10.5　优化

下面 OpenGLES_Ch10_1ViewController 的“-drawTerrainAndModels”方法的代码会在每次这个方法被调用时不加判断地绘制所有的地形瓦片和模型：

```
- (void)drawTerrainAndModels
{
   TETerrain *terrain = [[self dataSource] terrain];

if(nil == self.tiles)
{  // Cache tiles
   self.tiles = terrain.tiles;
}

// The terrain is opaque, so there is no need to blend.
glDisable(GL_BLEND);

[terrain drawTerrainWithinTiles:self.tiles
   withCamera:self.camera
   terrainEffect:self.terrainEffect];

// Assume subsequent rendering involves translucent objects.
glEnable(GL_BLEND);

// Configure modelEffect for texture and diffuse lighting
self.modelEffect.texture2D =
   [self.dataSource.modelManager textureInfo];
self.modelEffect.projectionMatrix =
   self.camera.projectionMatrix;
self.modelEffect.modelviewMatrix =
   self.camera.modelviewMatrix;

[self.modelEffect prepareToDraw];
```

```
    [self.dataSource.modelManager prepareToDraw];

    [terrain drawModelsWithinTiles:self.tiles
       withCamera:self.camera
       modelEffect:self.modelEffect
       modelManager:self.dataSource.modelManager];
}
```

例子 OpenGLES_Ch10_2 相比例子 OpenGLES_Ch10_1 做了一些温和、简单的改变，尽管如此，例子 OpenGLES_Ch10_2 仍然近乎加倍了每秒可以产生的帧数。例子 OpenGLES_Ch10_2 遵循了第 9 章的尽可能少渲染的忠告。基于平截体的剔除可以避免渲染看不到的瓦片。下面的 OpenGLES_Ch10_2ViewController 的“-drawTerrainAndModels”方法的实现可以提供一个更理想的结果，参见下面的粗体代码：

```
- (void)drawTerrainAndModels
{
   TETerrain *terrain = [[self dataSource] terrain];

   if(nil == self.tiles)
   {  // Cache tiles
      self.tiles = terrain.tiles;
   }

   NSMutableArray *fullDetailTiles = [NSMutableArray array];
   NSMutableArray *simplifiedTiles = [NSMutableArray array];

   [terrain identifyTilesToDraw:self.tiles
      withCamera:self.camera
      fullDetail:fullDetailTiles
      simplified:simplifiedTiles
      simplificationDistanceTiles:
         OpenGLES_Ch10_2DefaultSimplifaicationDistanceInTiles];

   // The terrain is opaque, so there is no need to blend.
   glDisable(GL_BLEND);

   [terrain drawTerrainWithinFullDetailTiles:fullDetailTiles
      simplifiedTiles:simplifiedTiles
      withCamera:self.camera
      terrainEffect:self.terrainEffect];

   // Assume subsequent rendering involves translucent objects.
   glEnable(GL_BLEND);
   // Configure modelEffect for texture and diffuse lighting
   self.modelEffect.texture2D =
      [self.dataSource.modelManager textureInfo];
   self.modelEffect.projectionMatrix =
```

```
        self.camera.projectionMatrix;
    self.modelEffect.modelviewMatrix =
        self.camera.modelviewMatrix;

    [self.modelEffect prepareToDraw];
    [self.dataSource.modelManager prepareToDraw];

    [terrain drawModelsWithinTiles:fullDetailTiles
        withCamera:self.camera
        modelEffect:self.modelEffect
        modelManager:self.dataSource.modelManager];
}
```

例子 OpenGLES_Ch10_2 会添加“-identifyTilesToDraw:withCamera:fullDetail: simplified:simplificationDistanceTiles:”方法到 TETerrain+viewAdditions.m 文件中。这个方法会遍历所有的地形瓦片并剔除看不到的瓦片，因为它们没有与当前摄像机的位置和方向所对应的视平截体相交。这个方法还会把可视的地形瓦片放在两个不同的数组中。接近摄像机位置的地形瓦片会放入要详细渲染的那个瓦片数组中。远处的瓦片会被放在要以简单的形式渲染的瓦片数组中。

```
////////////////////////////////////////////////////////////////////
// This method adds visible tiles from someTiles to either
// fullDetailTiles or simplifiedTiles. Tiles that are not visible
// by aCamera are not added to fullDetailTiles or simplifiedTiles.
//
// Visible tiles are added to simplifiedTiles if and only if
// the tile is greater than aNumberOfTiles from
// aCamera's position. Otherwise, visible tiles are added to
// fullDetailTiles.
- (void)identifyTilesToDraw:(NSArray *)someTiles
  withCamera:(UtilityCamera *)aCamera
  fullDetail:(NSMutableArray *)fullDetailTiles
  simplified:(NSMutableArray *)simplifiedTiles
  simplificationDistanceTiles:(GLfloat)aNumberOfTiles;
{
    const AGLKFrustum *frustumForCulling =
        [aCamera frustumForCulling];
    const GLfloat tileMetersPerUnit =
        self.metersPerUnit;
    const GLfloat tileWidthMeters = TETerrainTileDefaultWidth *
        tileMetersPerUnit;
    const GLfloat tileLengthMeters = TETerrainTileDefaultLength *
        tileMetersPerUnit;
    const GLfloat tileRadiusMeters = hypotf(tileWidthMeters,
        tileLengthMeters);
    const GLKVector3 cameraPositionMeters = aCamera.position;
    const GLfloat simplificationDistance =
```

```
      aNumberOfTiles * MAX(tileWidthMeters, tileLengthMeters);
   const GLfloat simplificationDistanceSquared =
      simplificationDistance * simplificationDistance;

   NSAssert(NULL != frustumForCulling, @"No valid frustum");

   for(TETerrainTile *tile in someTiles)
   {
      GLfloat originXMeters = tile.originX * tileMetersPerUnit;
      GLfloat originYMeters = tile.originY * tileMetersPerUnit;
      GLKVector3 centerMeters = GLKVector3Make(
         originXMeters + tileWidthMeters/2,
         0,
         originYMeters + tileLengthMeters/2);

      if(AGLKFrustumOut != AGLKFrustumCompareSphere(
         frustumForCulling, centerMeters, tileRadiusMeters))
      {  // Some part of the tile is visible
         if(simplificationDistanceSquared > distanceSquared(
            cameraPositionMeters, centerMeters))
         {  // Remember to draw the tile with full detail
            [fullDetailTiles addObject:tile];
         }
         else
         {  // Remember to draw the tile simplified
            [simplifiedTiles addObject:tile];
         }
      }
   }
}
```

在所有的瓦片中做剔除可以明显地改善渲染的速度，因为大部分的地形可以被剔除掉，并且在剔除的地形瓦片中的模型也会被剔除掉。即使在瓦片不能被整个剔除时，通常仍然可以用简化的几何图形来渲染它们。用户一般察觉不到在远处的瓦片中的任何可视细节的减少。

例子 OpenGLES_Ch10_2 会以三角形扇的形式绘制简化的瓦片，而不是三角形带。三角形扇是 glDrawArrays() 和 glDrawElements() 函数所支持的三个 OpenGL ES 三角形模式之一。GL_TRIANGLES 模式会让 OpenGL ES 使用所提供的每三个连续的顶点组来组成三角形。GL_TRIANGLE_STRIP 模式是在第 6 章介绍的，是渲染网格的一个最佳方式，具体是通过让每个三角形与前一个三角形共享两个顶点来组成三角形的。GL_TRIANGLE_FAN 模式是让所有的三角形共享一个“中心”点，并且让每个三角形与前一个三角形共享一个边缘顶点来组成三角形的。

图 10-7 展示了与地形瓦片的几何图形相似的三角形扇，但是每个边只有八个顶点。在图 10-7 中的箭头标示了 OpenGL ES 处理顶点的顺序。第一个顶点是“中心”点，但

它并不是必须要在这个图形的中心。在三角形扇中的所有三角形共享这个“中心”点。在图 10-7 中的三角形扇包含 28 个三角形，因为在这个图形中有 29 个顶点，但是第一个顶点是所有三角形共享的。

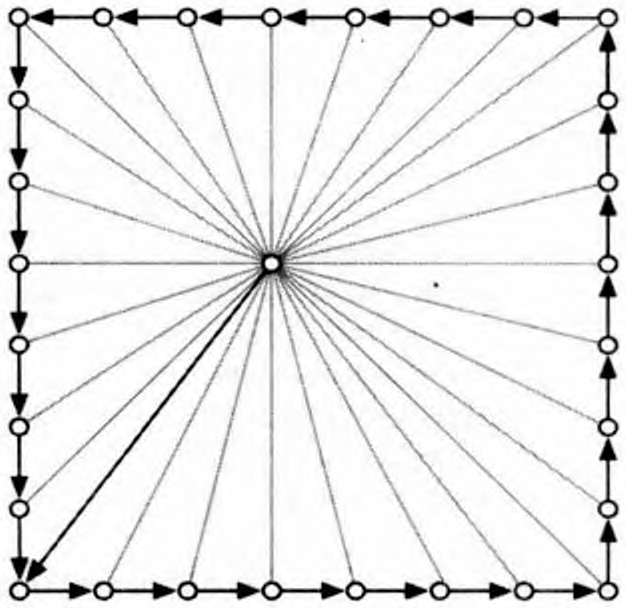


图 10-7　三角形扇是 OpenGL ES 内置三角形模式之一

使用 GL_TRIANGLE_FAN 模式来渲染简化的地形可以减少要渲染的三角形的数量。做一个比较，32×32 顶点的地形瓦片的网格拥有 31 行和 31 列。当以 GL_TRIANGLE_STRIP 模式渲染时，每个地形瓦片包含（31×31×2）+31=1953 个三角形，因为在每行每列中的每个矩形都是由两个三角形组成的，并且每行都包含一个“退化”的三角形，这个“退化”的三角形没有空间但是保留了顶点的处理顺序。与此相反，由地形瓦片的中心顶点和所有的边缘顶点形成的三角形扇只包含（4×31）=124 个三角形。瓦片有四个边，每个边有 31 行或者 31 列，所有三角形共享中心点。

下面的代码是 TETerrain+viewAdditions.m 文件中的“-drawSimplifiedTiles:”方法的实现，这个实现的工作方式与“-drawTiles:”方法的实现方式是相似的，除了其中的一行粗体代码。

```
//////////////////////////////////////////////////////////////////////
// This method submits vertex data to OpenGL for each tile in
// tiles. The pointer to the start of vertex data for each tile
// is set prior to rendering so that the vertex data for each
// tile is within accessible range of a single glDrawElements()
// call per tile. Tiles are drawn via TETerrainTile's
// -drawSimplified method.
- (void)drawSimplifiedTiles:(NSArray *)tiles;
{
   for(TETerrainTile *tile in tiles)
   {
      // Set the pointer to the first vertex position in the
      // tile
      glVertexAttribPointer(
         TETerrainPositionAttrib,
         3,
         GL_FLOAT,
         GL_FALSE,
         sizeof(GLKVector3),
         ((GLKVector3 *)NULL) +
            (tile.originY * self.width) +
            tile.originX);
```

```
            [tile drawSimplified];
        }
    }
```

在 TETerrainTile 类中的“-drawSimplified”方法的实现创建了一个包含在这个瓦片中的中心顶点和所有边缘顶点的顶点索引的元素数组缓存。接着使用 GL_TRIANGLE_FAN 模式调用 glDrawElements() 函数来渲染这个瓦片。

```
////////////////////////////////////////////////////////////////
// This method configures OpenGL ES state by binding element
// array buffers, and if necessary by passing element index data
// to the GPU. Then the vertices corresponding to the receiver
// are passed to the GPU via a call to glDrawElements(). The
// receiver's center and perimeter vertices are drawn as a
// triangle fan.
- (void)drawSimplified;
{
   if(0 == simplifiedIndexBufferID_ &&
      0 < [self.simplifiedIndexData length])
   {  // Indices haven't been sent to GPU yet
      // Create an element array buffer for mesh indices
      glGenBuffers(1, &simplifiedIndexBufferID_);
      NSAssert(0 != self.simplifiedIndexBufferID,
         @"Failed to generate element array buffer");
      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
         self.simplifiedIndexBufferID);
      glBufferData(GL_ELEMENT_ARRAY_BUFFER,
         [self.simplifiedIndexData length],
         [self.simplifiedIndexData bytes],
         GL_STATIC_DRAW);

      // No longer need local index storage
      self.simplifiedIndexData = nil;
   }
   else
   {
      glBindBuffer(GL_ELEMENT_ARRAY_BUFFER,
         self.simplifiedIndexBufferID);
   }

   glDrawElements(GL_TRIANGLE_FAN,
      self.numberOfSimplifiedIndices,
      GL_UNSIGNED_SHORT,
      ((GLushort *)NULL));
}
```

仔细检查在 TETerrainTile.m 文件中的“-generateIndicesForTileOriginX:tileOriginY:

tileWidth:tileLength:” 和 “-generateSimplifiedIndicesForTileOriginX:tileOriginY:tileWidth: tileLength:” 方法，学习一下这两个方法是怎么确定每个地形瓦片的详细的和简化的顶点索引的。这两个方法是相对简单的，仿效的是第 6.2.1 节中的例子。

OpenGLES_Ch10_2 中的优化只会影响两个类，TETerrainTile 和 TETerrain，并且这个优化会加倍显示刷新率。在第一代 iPad 上，例子 OpenGLES_Ch10_1 在渲染所包含的例子地形时可以在每秒内获得大约 14 帧（14 FPS）。例子 OpenGLES_Ch10_2 在渲染同一个地形时可以稳定地实现大约 30 帧。未优化前 iPad 2 可以实现大约 24 帧，优化后可以稳定地实现 50 帧，甚至更多。

其他潜在优化方式

大量学术论文探索了关于实时地形渲染的优化。专注游戏的网站 Gamasutra.com 为最著名的算法之一实时优化自适应网格（Real-time Optimally Adapting Mesh，ROAM，由 Mark Duchaineau 等人提出）提供了一个说明和示例代码，地址为：http://www.gamasutra.com/view/feature/3188/realtime_dynamic_level_of_detail_.php。

ROAM 算法可以动态自适应地形网格的细节级别，以用最少的三角形实现高视觉质量。但是渲染少量的不需要的三角形比让 CPU 去计算真正的最小数量的效率要更高。ROAM 的大部分实现还需要频繁地更新存储在 GPU 控制的内存中的顶点数据。当优化时，通常必须要考虑 CPU、GPU 和内存复制之间的平衡。例子 OpenGLES_Ch10_2 在 CPU 影响很小并且不用更新顶点数据的条件下提供了可接受的质量。

网页：http://vterrain.org/LOD/Papers/ 上有少量的论文和示例，它们介绍了一些 “超越” ROAM 的方法。当前研究探索的是使用运行在 GPU 上的 OpenGL ES Shading Language 程序来动态控制细节级别的问题。这些技术会把高度值编码到纹理中，利用 MIP 贴图来影响内置的用于选择顶点高度 GPU 纹理取样逻辑，甚至是在顶点间插补以提供比现存纹理更多的细节。MIP 贴图和纹理插补是在第 3 章讲解的。不幸的是，这个 Shading Language 方法要依赖于顶点程序的纹理取样能力。所有兼容 OpenGL ES 的 GPU 都支持片元程序内的纹理取样，但是只有很少支持顶点程序内的纹理取样，虽然 OpenGL ES 2.0 标准作为一个选项包含了这个能力。在例子 OpenGLES_Ch10_1 和例子 OpenGLES_Ch10_2 中的 UtilityTerrainEffect 类都是使用下面的语句来确定在顶点程序中可用的纹理取样单元的数量。

```
glGetIntegerv(GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS,
   &maxVertexTextureImageUnits);
```

不幸的是，在编写本书时没有一个在售的 iOS 设备支持顶点程序内的任何纹理取样单元。如果苹果公司提供改进的 OpenGL ES 驱动的话，其中一些 iOS 设备使用的 GPU 可能会支持顶点程序内的纹理取样。

10.6 小结

地形渲染为大型顶点数据集改进了网格渲染技术。瓦片集合会把地形顶点数据分割成足够小的适合 glDrawElements() 函数调用的块。这些瓦片还组成了一个简单的场景图，以实施类似第 9 章介绍的基于视平截体的剔除。

基于编码在帧缓存内的颜色信息的拾取实现了快速简单地确定用户触摸到的地形坐标的方法。Mac OS X 平台上的 TerrainEditor 应用选择放置在地形内的模型时也使用这种方法。基于颜色的拾取的优势是实现起来相对简单并且相对快速；以用于拾取的伪色彩来做渲染所要耗费的时间量与这个拾取所耗费的时间量是大体相同的，也就是说基于颜色的拾取所做的计算跟普通渲染相当。几乎所有的基于颜色的拾取的运算都是运行在 GPU 上的，而不是 CPU。尽管如此，其他的类似光线投射的更多依靠 CPU 的拾取方式可能在某些应用中工作得更好。例如，如果一个应用已经使用了一个物理引擎，那么使用基于光线的拾取可能就不需要额外的性能开销了。

TerrainEditor 应用会加载编码到图片文件中的像素亮度作为地形高度值。在例子 OpenGLES_Ch10_1 和 OpenGLES_Ch10_2 中的基于颜色的拾取过程中，会多次使用到可以与位置矢量（XYZ）相转换的颜色矢量（RGB）。地形的 X 和 Z 坐标被编码在一个 FBO 的附属像素颜色渲染缓存的 R 和 G 分量中。

第 11 章会回顾 3D 渲染所需的常见数学运算。以矢量表示的几何图形起初可能看似有悖常理，但是矢量运算几乎完全是由 GPU 能够快速、并行执行的加法和乘法组成的。第 11 章将为之前介绍的常见矢量运算提供简明的参考和解释。

第11章　数学速查

数学是计算机科学的基础，尤其是计算机图形学。线性代数包括了现代计算机图形中的大多数数学计算。矢量代数和矩阵数学也是关于这些运算的常见术语。对于矢量和矢量变换的研究是从17世纪开始的，在18世纪90年代成熟到现在的形式。查尔斯·巴贝奇在1837年设计了被称为“分析机”的多用途机械计算机。基于这个分析机，奥古斯塔·爱达·金，勒芙蕾丝伯爵夫人在1842年编写了第一个计算机程序。在1948年晶体管发明并替代了机械开关和真空管后，数学家们就已经知道可以用它们做什么了，然后软件产业诞生了。

计算机图形算法与雷达、蜂窝电话、视频编码和音乐合成中使用的数字信号处理算法是紧密相关的。例如，雷达分析反射的无线电波以确定三维空间中的一个物体的位置所使用的数学算法与找出用来描述一个3D场景中的对象的片元在像素颜色渲染缓存中的正确位置所使用的数学算法是相同的。本章介绍的运算适用于类似经济学、自然科学和社会科学的研究领域。抽象数学学习起来可能比较困难。幸运的是，对于大部分程序来说，只需要理解几个关键概念。本章会回顾前面章节逐渐介绍的线性代数概念，同时讲解关键概念和重要方程式，以及何时使用它们。

注意　经济学是线性代数的另一个常见用途：经济学中的很多问题都需要处理大量相互关联的方程式。在这些方程式中的变量可能是经济指标、货币供应量，或者生产率变动。对于线性代数，一个矩阵代表了一组值和另一组值之间的变换，一个经济体可以建模为一连串的变换。线性代数提供了管理数量庞大的相关因素和降低问题复杂性的工具。

11.1　概述

线性代数包括了对于矢量、坐标系（又被称为矢量空间）和函数的研究，其中函数会根据指定的线性方程组和矩阵系统的规则来变换矢量。几乎计算机图形的所有方面都

是使用线性代数实现的，虽然有类似三角学的替代方法可用。线性代数利用的是计算机能够快速执行的类似加法、减法和乘法的简单运算。

点、矢量和坐标轴等术语是在第 1 章介绍的。要理解本章的内容就有必要熟悉这些术语。回想第 1 章，矢量相加可以产生新的矢量。每个 3D 矢量可以描述为三个轴对齐矢量的和。3D 点的位置是使用三个坐标值 {X，Y，Z} 来存储的。一个点的位置同样可以表示为从坐标系原点到这个点的位置的一个矢量。

坐标系是使用矩阵定义的。使用矩阵可以把一个点的位置从一个坐标系高效地转换到另一个坐标系。最终，GPU 会要求所有的顶点位置以 OpenGL 默认坐标系来表达，以便 GPU 能够获悉点落在了渲染缓存的哪里。下一节会讲解矩阵是如何编码坐标系的定义的。

11.2 解码矩阵

所有可能存在的 3D 坐标系都能只使用 16 个值来编码。这些值被组织为内存中的 16 个连续的浮点值的数组，参见图 11-1。但是数学标记法通常会把一个 3D 矩阵表示为一个 4×4 的网格值矩阵。图 11-1 显示了内存中的值和数学标记法中的值之间的相互关系。

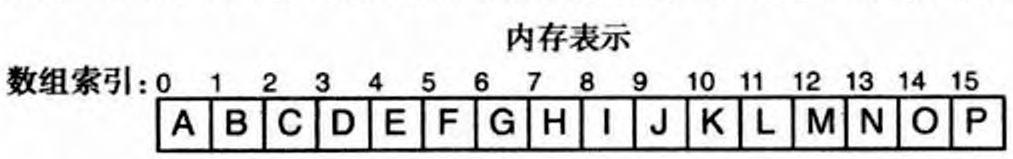


图 11-1　定义了一个 3D 坐标系的 4×4 矩阵

每个矩阵会编码三个单位矢量，这三个单位矢量描述了一个坐标系的正的 X、Y、Z 坐标轴的方向。第四个矢量确定了坐标原点的位置。图 11-2 标示了每个矢量。轴方向和原点的结合实际上定义了一个坐标系。

图 11-3 显示了一个“单位”矩阵。一个单位矩阵与任何其他矩阵级联（乘）的结果等于其他矩阵。单位矢量 {1.0，0.0，0.0} 定义了编码在单位矩阵中的 X 轴。单位矢量的长度通常等于 1.0。单位矢量 {0.0，1.0，0.0} 定义了编码在单位矩阵中的 Y 轴。单

位矢量 {0.0，0.0，1.0} 定义了编码在单位矩阵中的 Z 轴。单位矩阵的坐标轴原点在位置 {0.0，0.0，0.0} 处。

	X轴矢量	Y轴矢量	Z轴矢量	
	A	B	C	D
	E	F	G	H
	I	J	K	L
坐标轴原点	M	N	O	P

图 11-2　在一个矩阵中编码的矢量

	X轴矢量	Y轴矢量	Z轴矢量	
X分量	1.0	0.0	0.0	0.0
Y分量	0.0	1.0	0.0	0.0
Z分量	0.0	0.0	1.0	0.0
坐标轴原点	0.0	0.0	0.0	1.0

图 11-3　单位矩阵

矩阵定义了相对坐标系。你必须要回答这个问题，“相对于什么”，才能解释编码在矩阵中的值。图 11-4 显示了 OpenGL 默认坐标系是怎么与一个含有 3D 内容的全屏 Cocoa Touch UIView 对齐的。OpenGL 原点位于视图的中心。OpenGL 的 X 轴的方向是平行于视图的顶边和底边的，并且 X 轴的值是向右增加的。OpenGL 的 Y 轴是平行于视图的左边和右边的，并且 Y 轴的值向上增加。OpenGL 的 Z 轴值是沿着从视图指向观察者的一个矢量增加的。

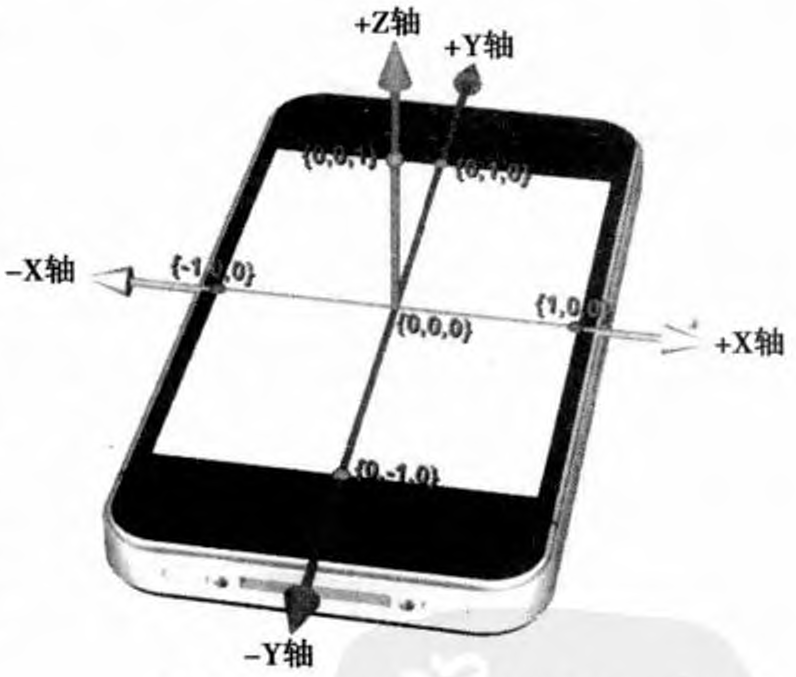
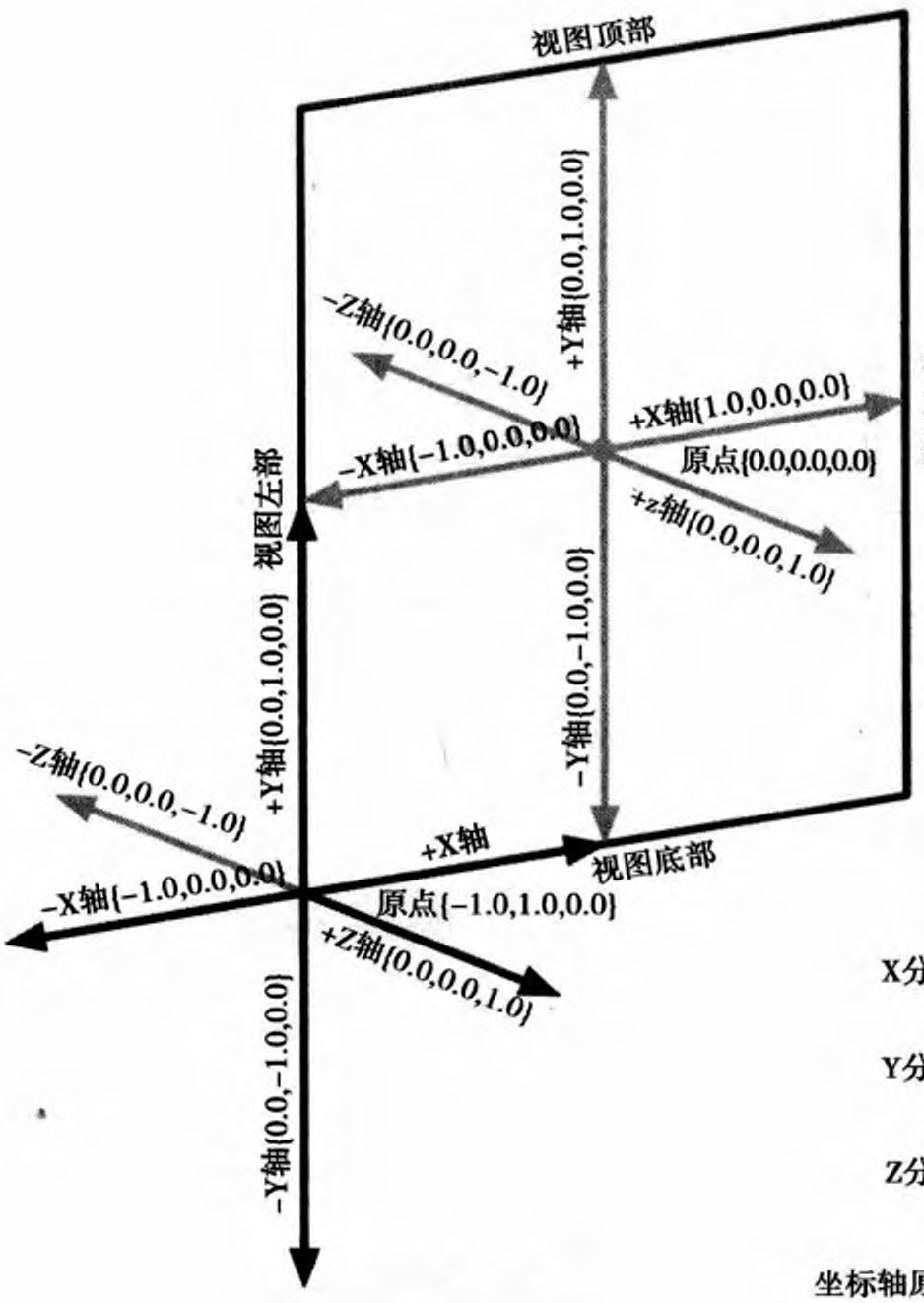


图 11-4　OpenGL 默认坐标系

OpenGL 应用会定义一个相对于默认坐标系的新坐标系。定义了一个原点在默认坐标系左下角的新 3D 坐标系的应用会创建了一个与图 11-5 中显示的类似的矩阵。图 11-5 的原点在默认坐标系的 {–1.0，–1.0，0.0} 位置处。

相对于另一个坐标系移动一个坐标系的原点叫做“平移”，这是第 5 章介绍的基本变换之一。平移只会影响一个矩阵的坐标轴“原点”部分。其他两个基本变换，旋转和缩放，只会影响编码在一个矩阵中的“轴矢量”。例如，旋转会相对于另一个坐标系改变每个轴矢量所指的方向。缩放会相对于另一个坐标系改变每个轴矢量的长度。图 11-6 展示了一个坐标系在经过平移、旋转和缩放后所发生的变化。例子 OpenGLES_Ch5_4 提供了一个用于帮助你测试和可视化坐标系变换的累加效果的用户界面。



	X轴矢量	Y轴矢量	Z轴矢量	
X分量	1.0	0.0	0.0	0.0
Y分量	0.0	1.0	0.0	0.0
Z分量	0.0	0.0	1.0	0.0
坐标轴原点	-1.0	-1.0	0.0	1.0

图 11-5　相对于 OpenGL 默认坐标系的一个平移坐标系

图 11-6　平移、旋转和缩放的图示

矩阵中的第四列是干什么的？截止到现在没有一个图片曾使用第四列来做任何事情。原来第四列只用在包含透视变换的矩阵中。OpenGL 一直保持了两个单独的矩阵，projectionMatrix 和 modelviewMatrix，部分是为了保持变换的编码简单性。按照约定，只有 projectionMatrix 会包含透视变换。当遵守这个约定时，在涉及 modelviewMatrix 的所有计算中都可以随意的忽略掉第四列。

注意　modelviewMatrix 和 projectionMatrix 总是会级联在一起创建一个单独的 modelview-ProjectionMatrix，以在 OpenGL ES 2.0 Shading Language 程序中使用。model-view 变换和投影变换之间的传统区别并不能表明它们之间有任何重要的数学区别。历史证明这个区别是很方便的，因为一些常见的计算只使用其中一个矩阵。例如，在灯光方程式中的法向量变换就需要一个继承自 modelviewMatrix 的特定矩阵。

11.2.1　从平截体获取矩阵

给定一个由视平截体定义的视点，可以很容易地构造出对应的 modelviewMatrix。视平截体是在第 8 章介绍过，并在第 9 章进一步讲解的。为了理解矩阵与平截体是怎么联系在一起的，请思考一下一个平截体是如何表示的。

从第 9 章开始，本书中的例子开始使用 AGLKFrustum 数据结构和用来操作这个数据结构的函数。尤其是 AGLKFrustumSetPositionAndDirection() 函数，这个函数会指定一个平截体的位置和方向。

```
void AGLKFrustumSetPositionAndDirection
(
 AGLKFrustum *frustumPtr,
 GLKVector3 eyePosition,
 GLKVector3 lookAtPosition,
 GLKVector3 upVector);
```

AGLKFrustumSetPositionAndDirection() 方法会计算从 eyePosition 到 lookAtPosition 的视线方向。这个平截体的 Z 轴矢量是一个指向视线方向的单位矢量。

```
// Compute the Z axis of the frustum. The Z axis points in
// the direction from eye position to look at position
const GLKVector3 lookAtVector =
   GLKVector3Subtract(eyePosition, lookAtPosition);
   frustumPtr->zNormalVector = GLKVector3Normalize(lookAtVector);
```

任意两个非平行单位矢量的矢量积会产生第三个与前两个矢量垂直的单位矢量。矢量积是在第 4 章介绍的，11.4.3 节会再做介绍。

给定一个上方向和一个视线方向，那么平截体的 X 轴矢量是“上”矢量和“视线”矢量的矢量积。图 11-7 显示了上方向、视线方向和计算出来的平截体 X 轴矢量之间的关系。

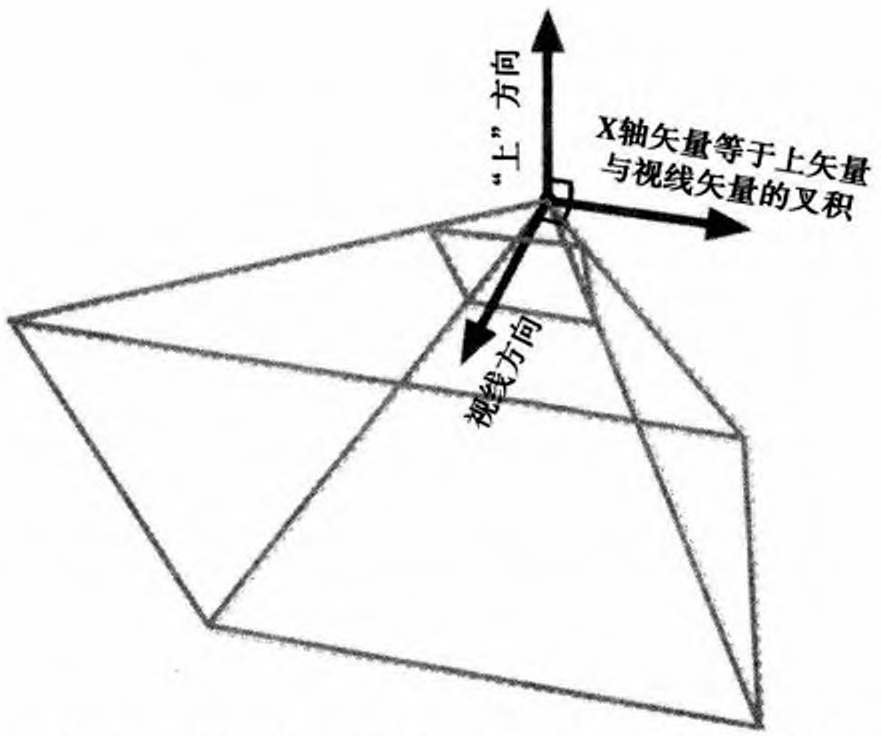


图 11-7 “视线”和“上”矢量确定了平截体的方位

平截体的 X 轴是视线矢量和上矢量之间的标准化矢量积。如下代码中的 zNormal-Vector 是标准化视线矢量。

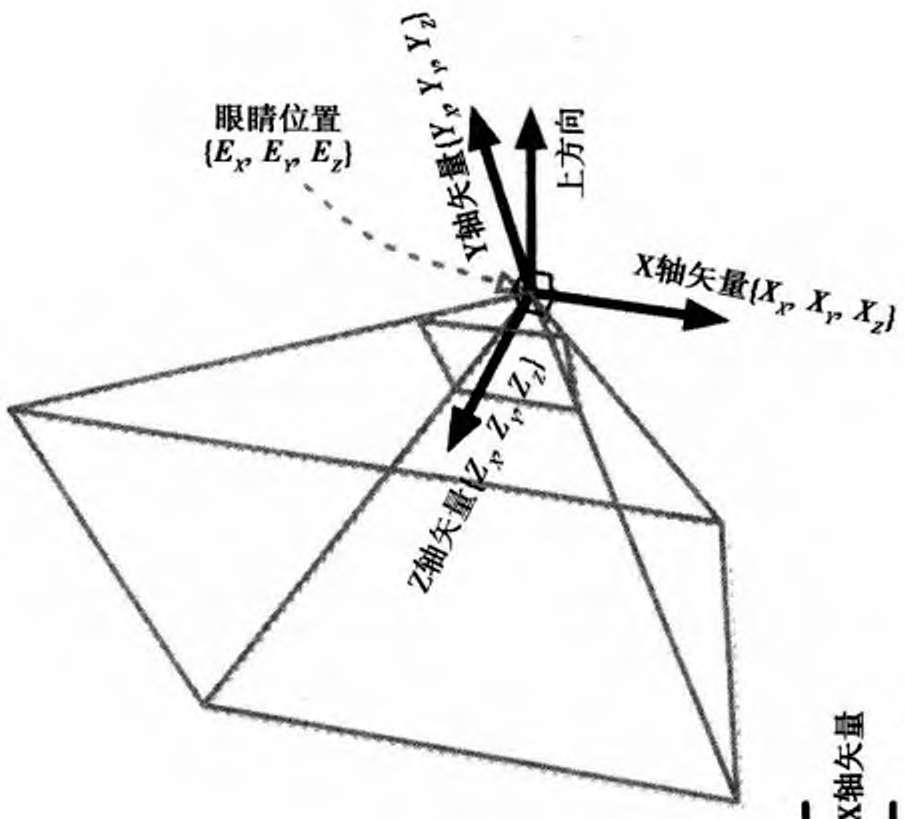
```
// The frustum's X normal is the cross product of the
// normalized "up" vector and the frustum's Z normal
frustumPtr->xNormalVector = GLKVector3CrossProduct(
   GLKVector3Normalize(upVector),
   frustumPtr->zNormalVector);
```

传递给 AGLKFrustumSetPositionAndDirection() 函数的 upVector 不能总是不加更改地作为平截体的 Y 轴，因为这个 upVector 可能不是一个单位矢量，而且可能并不与 X 轴和 Z 轴都垂直。平截体的真正 Y 轴是通过计算 Z 轴和 X 轴的矢量积得到的。

```
// The frustum's Y axis is the cross product of the
// frustum's Z normal and the frustum's X normal.
frustumPtr->yNormalVector = GLKVector3CrossProduct(
   frustumPtr->zNormalVector,
   frustumPtr->xNormalVector);
```

图 11-8 显示了平截体的轴矢量和对应矩阵之间的关系。这个平截体的坐标轴提供了对应于这个平截体的“视线”model-view 矩阵中的轴矢量。图 11-8 中的坐标轴原点源自这个平截体的 eyePosition。

下面的 GLKMatrix4 AGLKFrustumMakeModelview(const AGLKFrustum *frustumPtr) 函数会返回一个编码了指定平截体的位置和方向的 modet-view 矩阵。



	X轴矢量	Y轴矢量	Z轴矢量	
	X_x	Y_x	Z_x	0.0
	X_y	Y_y	Z_y	0.0
	X_z	Y_z	Z_z	0.0
坐标轴原点=眼睛位置	$-E_x$	$-E_y$	$-E_z$	1.0

图 11-8　与一个平截体对应的轴矢量

```
//////////////////////////////////////////////////////////////////////
// Returns a modelview matrix that encodes the point of view
// matching the specified frustum
GLKMatrix4 AGLKFrustumMakeModelview
(
 const AGLKFrustum *frustumPtr)
{
   NSCAssert(AGLKFrustumHasDimention(frustumPtr),
      @"Invalid frustumPtr parameter");

   const GLKVector3 eyePosition = frustumPtr->eyePosition;
   const GLKVector3 xNormal = frustumPtr->xNormalVector;
   const GLKVector3 yNormal = frustumPtr->yNormalVector;
   const GLKVector3 zNormal = frustumPtr->zNormalVector;
   const GLfloat xTranslation = GLKVector3DotProduct(
      xNormal, eyePosition);
   const GLfloat yTranslation = GLKVector3DotProduct(
      yNormal, eyePosition);
```

```
    const GLfloat zTranslation = GLKVector3DotProduct(
       zNormal, eyePosition);

    GLKMatrix4 m = {
       // X Axis     Y Axis     Z Axis
       xNormal.x, yNormal.x, zNormal.x,            0.0f,
       xNormal.y, yNormal.y, zNormal.y,            0.0f,
       xNormal.z, yNormal.z, zNormal.z,            0.0f,

       // Axis Origin
       -xTranslation, -yTranslation, -zTranslation, 1.0f
    };

    return m;
}
```

一个平截体的 model-view 矩阵的平移可以表示为这个平截体的定义所在的坐标系的位移。图 11-5 是对这个关系的最简单的可视化说明。换句话说，一个坐标系的原点被定义为另一个坐标系中的一个位置，另一个坐标系指的是新坐标系的相对坐标系。平移会移动一个坐标系的原点。平移量是在 AGLKFrustumMakeModelview() 函数中反求出来的，因为要把上一个坐标系中的眼睛位置转换成新坐标系的原点 {0，0，0} 位置就有必要减去眼睛位置的每个分量。

11.2.2 透视

就像 AGLKFrustumSetPositionAndDirection() 函数会设置一个平截体的位置和方向，AGLKFrustumSetPerspective(AGLKFrustum *frustumPtr, GLfloat fieldOfViewRad, GLfloat aspectRatio, GLfloat nearDistance, GLfloat farDistance) 函数用于定义一个平截体的形状。图 11-9 显示了 AGLKFrustumSetPerspective() 参数与结果平截体之间的几何关系。

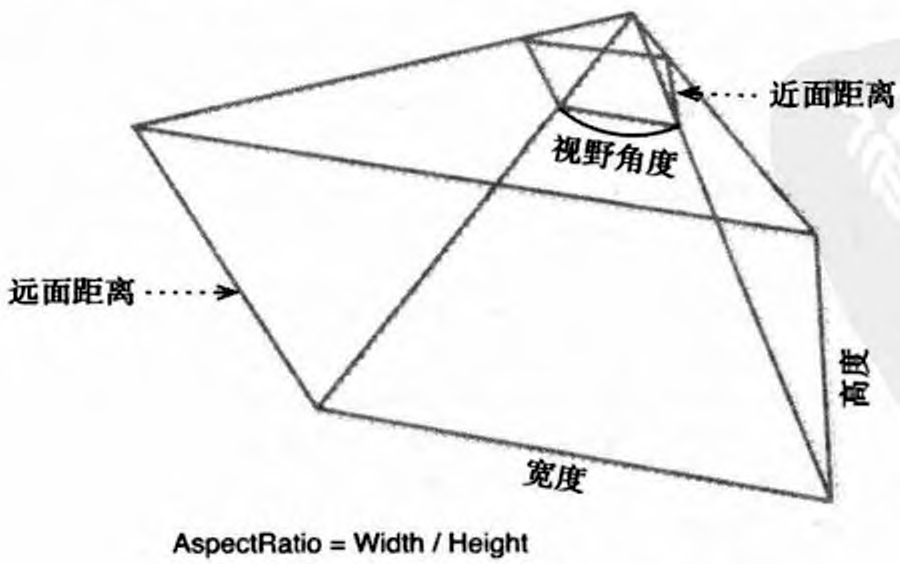


图 11-9 定义一个平截体的形状

这个平截体的形状确定了与这个平截体对应的“透视”投影矩阵。下面的 GLKMatrix4 AGLKFrustumMakePerspective(const AGLKFrustum *frustumPtr) 函数会返回与这个平截体对应的一个投影矩阵。

```
//////////////////////////////////////////////////////////////////////
// Returns a projection matrix that encodes perspective
// matching the specified frustum
extern GLKMatrix4 AGLKFrustumMakePerspective
(
 const AGLKFrustum *frustumPtr)
{
   NSCAssert(AGLKFrustumHasDimention(frustumPtr),
      @"Invalid frustumPtr parameter");
   const GLfloat cotan =
      1.0f / frustumPtr->tangentOfHalfFieldOfView;
   const GLfloat nearZ = frustumPtr->nearDistance;
   const GLfloat farZ = frustumPtr->farDistance;

   GLKMatrix4 m = {
      cotan / frustumPtr->aspectRatio, 0.0f, 0.0f, 0.0f,
      0.0f, cotan, 0.0f, 0.0f,
      0.0f, 0.0f, (farZ + nearZ) / (nearZ - farZ), -1.0f,
      0.0f, 0.0f, (2.0f * farZ * nearZ) / (nearZ - farZ), 0.0f
   };

   return m;
}
```

在 AGLKFrustumMakePerspective() 函数中计算的 1/2 视野角度的余切值，代表了一个物体的宽度和高度与这个物体跟坐标系原点之间的距离的关系。这个余切值控制了透视图是怎么让远处的物体比近处的物体显得更小的。减小视野角度可以获得放大物体的视觉效果，从而使它们看起来更近一些，就好像是通过望远镜来看它们。

11.2.3　矢量的坐标轴分量

两个矢量的标量积会提取每个矢量在另一个矢量方向上的分量。标量积是在第 4 章介绍的。假设一个由两个矢量（矢量 A 和矢量 B）定义的三角形，参见图 11-10。

遵循直角三角形的标准命名约定，矢量 A 是这个三角形的“临边”，矢量 B 是这个三角形的斜边。对于一个角度为 θ 的直角三角形来说，角度 θ 的余弦值等于临边长除以斜边长。非常方便的是，标量积运算会返回相同的值，即临边长除以斜边长所得的值，不需

图 11-10　标量积的图示

要任何三角学知识。

图 11-11 提供了可视化两个矢量的标量积的另一种方式。在图 11-11 中，单位矢量 A 与 X 轴单位矢量的标量积提供了单位矢量 A 在 X 轴方向上的长度部分。单位矢量 A 与 Y 轴单位矢量的标量积提供了单位矢量 A 在 Y 轴方向上的长度部分。

获得一个矢量在另一个矢量方向上的长度的能力有无数的用途。请记住标量积运算适用于任意两个矢量。实际上这个运算是没有必要包含一个坐标轴矢量的。例如，图 11-12 使用一条虚线图示了单位矢量 A 在单位矢量 B 方向上的部分。换句话说，在图 11-12 中显示的虚线与单位矢量 B 的交点指示了单位矢量 A 与单位矢量 B 的标量积。

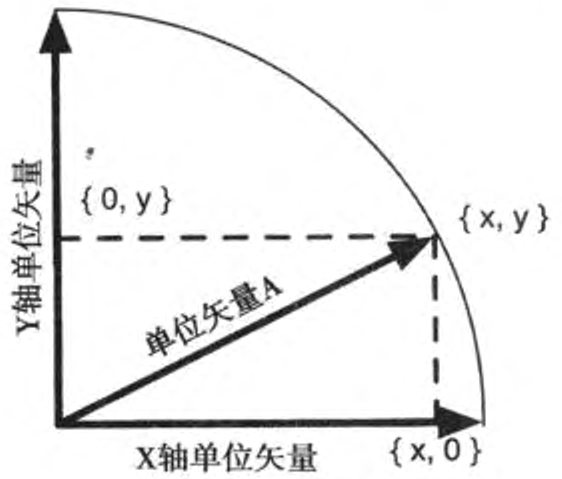


图 11-11 一个矢量在另一个矢量的方向上的长度

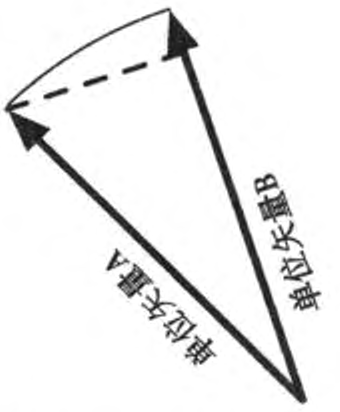


图 11-12 标量积适用于任意两个单位矢量

在 GLKit 的 GLKVector3.h 文件中，苹果公司提供了计算两个 3D 矢量的标量积的 C 代码。这个代码看起来会与下面的代码相似。其中标量积的计算只使用了乘法和加法，而这两种运算是 GPU 执行最快的运算。

```
float GLKVector3DotProduct(
   GLKVector3 vectorLeft,
   GLKVector3 vectorRight)
{
    return vectorLeft.x * vectorRight.x +
           vectorLeft.y * vectorRight.y +
           vectorLeft.z * vectorRight.z;
}
```

11.2.4 点变换

当你有了一个矩阵后，可以用来干什么呢？矩阵可以用于计算与另一个坐标系中的同一个点的位置对应的一个点在一个坐标系中的位置。从一个坐标系向另一个坐标系转换一个点的位置的操作被称为“变换”或者“投影”。两个术语是可互换的。换句话说，任意一个点都可以基于各种坐标系定义。在一个坐标系的 {55.5，11.0，100.2} 位置处的点可能在另一个坐标系的 {10.0，–100.0，20.0} 位置处。

回想一下，任意一个点的位置都可以表示为从坐标系原点到这个点的一个矢量。要变换一个矢量，请用一个矩阵乘以这个矢量。结果是与这个矩阵相关的坐标系内的相同矢量。本书中的所有 iOS 例子几乎从未使用 Objective-C 或者 C 代码来变换矢量，因为这个运算 GPU 执行得最好。本书中的很多 OpenGL ES 2.0 Shading Language 例子会包含“gl_Position = u_mvpMatrix * vec4(a_position, 1.0);”语句，一个位置乘以 modelviewProjectionMatrix（mvpMatrix）矩阵的结果是在 OpenGL 默认坐标系中表示的同一个位置。

如果你需要用 C 代码来变换一个矢量，苹果公司的 GLKit 框架提供了函数 GLKMatrix4MultiplyVector3(GLKMatrix4 matrixLeft, GLKVector3 vectorRight) 和 GLKMatrix4MultiplyVector3WithTranslation(GLKMatrix4 matrixLeft, GLKVector3 vectorRight)。函数 GLKMatrix4MultiplyVector3() 会执行变换，并忽略编码在这个矩阵的坐标轴原点部分中的任意平移。GLKMatrix4MultiplyVector3WithTranslation() 函数会在变换中包括平移。问题是没有任何数学上定义的运算能够把一个 4×4 矩阵乘以一个三分量矢量。这两个 GLKit 函数都会把 3D 矢量扩展为一个四分量矢量。GLKMatrix4MultiplyVector3 函数会使用 0 来扩展矢量以转换 3D 矢量 {x，y，z}，为 4D 矢量 {x，y，z，0.0}。当用 0 分量乘以编码在这个矩阵的坐标轴原点部分中的平移值时，平移就会无效。GLKMatrix4MultiplyVector3WithTranslation() 函数会扩展 3D 矢量为一个 4D 矢量 {x，y，z，1.0}。当这个 4D 矢量乘以这个矩阵时，编码在这个矩阵的坐标轴原点部分中的平移值会乘以 1.0，并且因此会包含平移转换。

你可以在苹果公司的 GLKit 的 GLKMatrix4.h 文件中看到实现 4D 矢量变换的 C 代码。下面的函数实现突出显示了用来计算变换矢量的每个分量的运算代码。

```
GLKMatrix4 GLKMatrix4MultiplyVector4(
   GLKMatrix4 matrix,
   GLKVector4 vector)
{
   GLKVector4 v = {
      matrix.m[0]  * vector.x +
      matrix.m[4]  * vector.y +
      matrix.m[8]  * vector.z +
      matrix.m[12] * vector.w,

      matrix.m[1]  * vector.x +
      matrix.m[5]  * vector.y +
      matrix.m[9]  * vector.z +
      matrix.m[13] * vector.w,

      matrix.m[2]  * vector.x +
      matrix.m[6]  * vector.y +
      matrix.m[10] * vector.z +
      matrix.m[14] * vector.w,
```

```
        matrix.m[3]  * vector.x +
        matrix.m[7]  * vector.y +
        matrix.m[11] * vector.z +
        matrix.m[15] * vector.w
    };

    return v;
}
```

在 GLKMatrix4MultiplyVector4() 函数中的计算看起来很熟悉，这并非巧合。它们执行的是四维的标量积计算。使用一个矩阵变换一个点会提取从这个坐标系原点到这个点的矢量的分量。这些目标分量是与编码在这个矩阵中的每个坐标轴对齐的分量。变换的 X 坐标是在这个矩阵的 X 轴方向上的矢量部分。变换的 Y 坐标是在这个矩阵的 Y 轴方向上的矢量部分。变换的 Z 坐标是在这个矩阵的 Z 轴方向上的矢量部分。在 GLKMatrix4MultiplyVector4() 函数的实现中的粗体代码突出了四分量矢量的最后一个 w 分量与编码在矩阵坐标轴原点的平移值之间的相互作用。

11.2.5 转置矩阵和逆矩阵

矩阵定义了一个坐标系与另一个之间的关系。这个关系可以翻转。一个把坐标系 A 映射到坐标系 B 的矩阵可以被转置为映射坐标系 B 到坐标系 A。转置一个矩阵是通过切换一个矩阵的行和列来完成的。图 11-13 描述了一个矩阵以及这个矩阵对应的转置矩阵。

请注意在图 11-13 中沿着矩阵 A 对角线的阴影矩阵元素与矩阵 A 转置矩阵对角线位置的元素是相同的。思考转置操作的一个方法是把转置矩阵看做是源矩阵的一个倒影，与源矩阵沿对角线对称。还要注意的是转置矩阵的转置会再次产生源矩阵，在图 11-13 中矩阵 A 的转置的转置会再次产生矩阵 A。

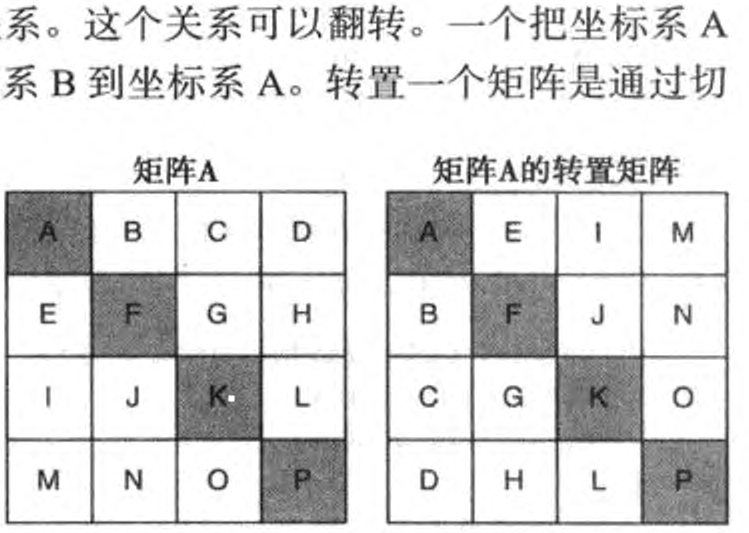


图 11-13 一个矩阵以及这个矩阵对应的转置矩阵

GLKit 提供了用于返回矩阵的转置的 GLKMatrix4 GLKMatrix4Transpose(GLKMatrix4 matrix) 函数。

一个矩阵的逆是与源矩阵相乘会产生单位矩阵的矩阵。给定矩阵 A 和它的逆——矩阵 B，那么矩阵 A 乘以矩阵 B 会产生单位矩阵。对于常见矩阵来说，计算矩阵的逆需要做很多工作。但是，当一个矩阵的所有坐标轴被施加了一致的缩放时，矩阵的逆就等于矩阵的转置。并不是所有的矩阵都是可逆的。例如，如果一个矩阵包含一个值为 0 的缩放因素，那么这个矩阵就没有逆矩阵。

GLKit 提供了 GLKMatrix4 GLKMatrix4Invert(GLKMatrix4 matrix, bool *isInvertible) 函数。其中的 isInvertible 参数会返回一个布尔值，来告诉你这个矩阵是否可逆。

GLKit 还提供了 GLKMatrix4InvertAndTranspose(GLKMatrix4 matrix, bool *isInvertible) 方法，这个方法会返回矩阵的逆的转置（如果有的话）并会通过 isInvertible 参数告诉你这个矩阵是否可逆。当使用标准 OpenGL 灯光方程时，基于法向量的灯光的正确计算需要用到 model-view 矩阵的逆的转置。GLKBaseEffect 类会按需计算 model-view 矩阵的逆的转置以返回 baseeffect 的 normalMatrix 属性。在例子 OpenGLES_Ch7_3 中的 UtilityArmatureBaseEffect 类和第 10 章例子中的 UtilityModelEffect 类都会使用 GLKMatrix4InvertAndTranspose() 方法来计算一个 mormalMatrix 属性。这些例子间接地展示了在 OpenGL ES Shading Language 程序中的灯光方程是怎么使用 normalMatrix 的。

11.3　四元法

四元法是爱尔兰数学家 William Rowan Hamilton 先生命名的。Hamilton 的突破是在 1843 年 10 月 16 日，星期一，在都柏林。当他跟他的妻子沿着皇家运河纤道散步时，突然想到了四元法的概念。为了把这个概念留给后人，在路过布鲁穆桥时，他把四元法的公式刻在了桥上。这个雕刻作为一个历史地标一直保留至今。

GLKit 包含了 GLKQuaternion 数据类型和用来操作四元法的函数。每个四元数由一个矢量 v 和一个标量值 s 组成，这个 s 指定了绕着 v 的旋转量。3D 图形应用需要使用四元法，因为当使用四元法时，可以实现绕着坐标轴的任意组合做任何旋转，而且可以避免可怕的“万向节锁”。例如，函数 GLKMatrix4MakeLookAt() 的实现所使用的数学计算无法产生一个直接沿着“上”矢量看的有效视点。这个限制的存在是因为当直接看向“上”或者“下”时，GLKMatrix4MakeLookAt() 函数中的数学计算会试图计算 90 度的正切，而这在数学上是未定义的。这个“未定义”的现象也会在真实世界中发生，就是当机械万向节碰到“万向节锁”和捕捉或者产生了摇摆不稳定的行为时。维基百科上有一个可视化万向节锁的动画，地址为：http://en.wikipedia.org/wiki/Gimbal_lock。

四元数和矩阵之间的转换是双向的。函数 GLKMatrix4 GLKMatrix4MakeWithQuaternion(GLKQuaternion quaternion) 会返回一个矩阵，这个矩阵所包含的旋转跟参数 quaternion 所包含的旋转是相同的。GLKQuaternion GLKQuaternionMakeWithMatrix4(GLKMatrix4 matrix) 函数会返回一个四元数，这个四元数编码了 matrix 参数所包含的所有旋转。这个来回转换的能力提供了灵活性。本书中没有例子使用过四元数，不过当你熟悉了本书中的其他概念后，可以把 OpenGLES_Ch6_1 和 OpenGLES_Ch7_1 中的碰碰车例子转换成使用四元数的有趣的高级工程。很多情况下使用四元法可以简化你的代码。例如，在多个第三方 3D 游戏框架中使用了基于四元法的摄像机类。

网址：http://www.cprogramming.com/tutorial/3d/quaternions.html 中有一个关于四元法的简明教程。GLKit 的 GLKQuaternion.h 文件揭示了苹果公司是怎么使用内联 C 代码的形式实现四元法的。虽然其中的数学计算超出了本书的讲解范围，但还算是比较简单

的。四元法是使用 4D 空间来描述 3D 角度。如你所想，用文字讲解和描述四维是一个相当大的挑战。基于这个原因，更好地学习资源应该是四元数的在线演示和说明。

11.4 常用的图形数学

本章是 3D 图形数学的一个速成课。如果本章让你头晕，不要太担心。本节列出了应用会执行的最常见的 3D 数学运算。使用这些诀窍和技术可以解决你所碰到的大部分问题，甚至不必深入理解其实现。

11.4.1 简单矢量运算

下面是常见的矢量运算。

- 翻转方向。GLKVector3 GLKVector3Negate(GLKVector3 vector) 函数会返回一个跟 vector 参数长度相同但是方向相反的新矢量。你可以在第 6 章的 SceneCar 类中看到这个函数。
- 长度缩放。GLKVector3 GLKVector3MultiplyScalar(GLKVector3 vector, float value) 函数会使用 value 参数的值缩放 vector 参数的长度，但是不会改变 vector 参数的方向，除非值为负。函数调用 GLKVector3MultiplyScalar(vect-or, -1.0) 和 GLKVector3Negate(vector) 返回的是相同的矢量。在第 4 章使用 GLKVector3-MultiplyScalar() 平衡表面法向量以获得平滑的灯光效果。SceneCar 例子类、AGLKit 例子类和 Utility 例子类也使用了这个函数。
- 两点间矢量。GLKVector3 GLKVector3Subtract(GLKVector3 position1, GLKVector3 position2) 函数会返回从 position2 到 position1 之间的矢量。第 4 章在计算用于灯光模拟的表面法向量过程的一个步骤中计算了顶点间的矢量。多个 AGLKit 和 Utility 例子类使用了这个函数。
- 矢量相加。GLKVector3 GLKVector3Add(GLKVector3 vectorA, GLKVector3 vectorB) 函数会返回一个相当于在 vectorA 方向上移动 vectorA 长度，并在 vectorB 方向上移动 vectorB 长度的一个新矢量，参见第 1 章的图 1.11。第 4 章在为平滑灯光模拟而平衡表面法向量的过程中应用了矢量相加。
- 矢量标准化。GLKVector3 GLKVector3Normalize(GLKVector3 vector) 函数会返回一个跟 vector 参数方向相同，但是长度为 1.0 的新矢量。长度为 1.0 的矢量被称为单位矢量。第 4 章讲解了法向量。
- 两点间距离。float GLKVector3Distance(GLKVector3 positionA, GLKVector3 positionB) 函数会返回参数 positionA 和 positionB 之间的距离。它相当于 float GLKVector3-Distance(GLKVector3 positionA, GLKVector3 positionB)。第 6 章的 SceneCar 类调用了 GLKVector3Distance() 函数。

11.4.2　矢量标量积

本章的第 11.2.3 节展示了标量积函数 float GLKVector3DotProduct(GLKVector3 vectorA, GLKVector3 vectorB) 的实现。标量积有以下常见用途。

- 矢量间角度。标量积是两个矢量间夹角的余弦值。因此 acosf(GLKVector3DotProduct(vectorA, vectorB)) 得出的是两个矢量间的角度。第 6 章的 SceneCar 类调用了 GLKVector3DotProduct() 函数，用以决定把碰碰车模型掉转到它正移动的方向需要旋转多少度。
- 计算长度的平方。一个矢量与它自身的标量积等于这个矢量长度的平方。很多情况下要用到长度的平方，因为如果 lengthA 的平方大于 lengthB 的平方，那么 lengthA 也大于 lengthB。常常只需要知道哪一个更长而不需要知道精确的长度值。可参看在第 10 章的例子 TerrainEditor 中的 UtilityVector3LengthSquared() 函数的实现。
- 长度计算。表达式 sqrtf(GLKVector3DotProduct(vectorA, vectorA)) 会计算 vectorA 的长度，这个表达式相当于 float GLKVector3Length(GLKVector3 vectorA) 函数。sqrtf() 函数是标准 C 库中计算代价最高的函数之一。任何时刻都应该尽量避免调用它。在很多情况下都可以使用一个矢量长度的平方来代替实际长度。第 6 章中的 SceneCar 类的实现代码中调用了 GLKVector3Length() 函数，第 10 章的例子使用这个函数来控制摄像机的移动。
- 确定视点背后的点。如果表达式 GLKVector3DotProduct(GLKVector 3Subtract (position, eyePosition), lookAtVector)) 的值为负，那么 position 位于 eyePosition 的后面，因此无法被看到。这个技术是在第 8 章介绍的，用以避免绘制那些看不到的公告牌特效。
- 计算漫反射光。射向一个三角形的漫反射光的强度等于表达式 GLKVector3DotProduct(diffuseLightDirection, surfaceNormalvector)) 的值。第 4 章讲解了标量积在灯光模拟中的用途。
- 分离矢量分量。表达式 GLKVector3DotProduct(vectorA, vectorB) 会返回 vectorA 在 vectorB 方向上的分量部分。表达式 GLKVector3DotProduct (vectorA, zAxisVector) 会返回 vectorA 在 Z 轴方向上的分量。GLKVector3DotProduct (vectorA, yAxisVector) 会返回 vectorA 在 Y 轴方向上的分量。表达式 GLKVector3DotProduct(vectorA, xAxisVector) 会返回 vectorA 在 X 轴方向上的分量。请参看在第 9 章中介绍的 AGLKFrustum 例子类以及 11.2.3 节。

11.4.3　矢量的矢量积

两个非平行单位矢量的矢量积是垂直于这两个非平行单位矢量的第三个单位矢量，参见图 11-14。

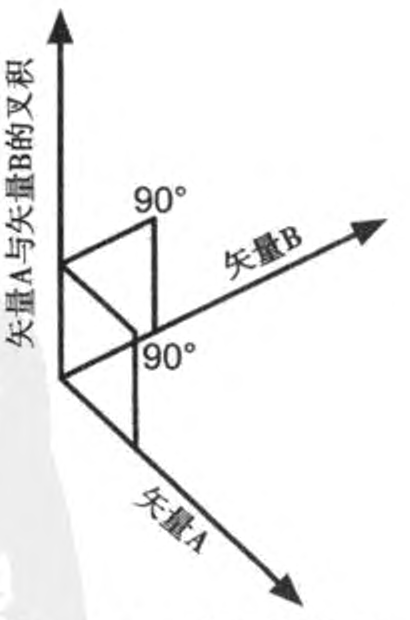


图 11-14　标量积的图示

矢量积的计算只需要乘法和加法运算，这两个计算是 GPU 执行最快的运算。

```
GLKVector3 GLKVector3CrossProduct(
   GLKVector3 vectorLeft,
   GLKVector3 vectorRight)
{
    GLKVector3 v = {
      vectorLeft.y * vectorRight.z - vectorLeft.z * vectorRight.y,
      vectorLeft.z * vectorRight.x - vectorLeft.x * vectorRight.z,
      vectorLeft.x * vectorRight.y - vectorLeft.y * vectorRight.x };

    return v;
}
```

- 表面法向量。表面法向量是垂直于由两个矢量所定义的面的一个单位矢量。GLKVector3CrossProduct(GLKVector3Normalize (vectorA), GLKVector3-Normalize(vectorB)) 函数会返回由 vectorA 和 vectorB 所定义的面的表面法向量。第 4 章讲解了用于灯光模拟的表面法向量的计算。第 12 章会使用相同的计算以确保装载了火箭发动机的碰碰车模型在驶过山谷和丘陵时会与地形保持平行。
- 平截体几何图形。一个平截体的视线方向矢量和上方向矢量之间的矢量积产生了这个平截体的 X 轴矢量。第 8 章和第 9 章讲解了一个 3D 场景中的可视区域的几何定义。11.2.1 节介绍了一个平截体的几何定义与对应的矩阵表示之间的关系。
- 公告牌方向。公告牌是与平截体的近面平行的。一个平截体的视线方向矢量和上方向矢量的矢量积乘以公告牌宽度的一半确定了公告牌左下角的位置。如第 8 章介绍的，其余的公告牌角是相对于左下角计算出来的。

11.4.4 model-view 矩阵

model-view 矩阵定义了用来描述一个场景内的物体的位置和方向的坐标系。model-view 矩阵定义了视平截体的位置和方向。

- 初始化一个 model-view 矩阵。初始化一个 model-view 矩阵的最常见方式是调用 GLKMatrix4 GLKMatrix4MakeLookAt(float eyeX, float eyeY, float eyeZ, float centerX, float centerY, float centerZ, float upX, float upY, float upZ) 函数。model-view 矩阵还可以由一个视平截体或者一个四元数计算而来。
- 与默认坐标系之间的关系。可以使用 model-view 矩阵把一个点从 model-view 坐标系变换到投影坐标系。model-view 矩阵和投影矩阵的级联产生了一个单独的矩阵，这个矩阵可以把一个点从 model-view 坐标系一直变换到 OpenGL 默认坐标系。
- 级联。两个矩阵的级联会产生第三个包含了这两个矩阵的所有坐标系变换的矩阵。函数 GLKMatrix4 GLKMatrix4Multiply(GLKMatrix4 matrixA, GLKMatrix4

matrixB) 会返回 matrixA 和 matrixB 的级联。

- 累积变换。model-view 矩阵通常包含与透视图无关的所有坐标系变换。函数 GLKMatrix4 GLKMatrix4Scale(GLKMatrix4 matrix, float sx, float sy, float sz) 会返回，通过使用指定的因数缩放编码在矩阵中的坐标轴所产生的一个新矩阵。函数 GLKMatrix4 GLKMatrix4Translate(GLKMatrix4 matrix, float tx, float ty, float tz) 会返回，通过使用指定的轴对齐距离来平移编码在矩阵中的坐标轴原点所产生的一个新矩阵。函数 GLKMatrix4 GLKMatrix4RotateWithV ector3(GLKMatrix4 matrix, float radians, GLKVector3 axisVector) 会返回，通过使用指定的围绕参数 axisVector 的弧度值来旋转编码在矩阵中的坐标轴所产生的一个新矩阵。

11.4.5　投影矩阵

投影矩阵要么会定义一个正交投影，要么会定义一个透视投影。正交投影不会基于物体与坐标系原点的距离而改变物体的显示大小。透视投影会让离坐标系原点较远的物体显得更小。投影矩阵定义了视平截体的形状，投影矩阵通常是通过调用 GLKMatrix4 GLKMatrix4 MakePerspective(float fovyRadians, float aspect, float nearZ, float farZ) 或者 GLKMatrix4 GLKMatrix4MakeOrtho(float left, float right, float bottom, float top, float nearZ, float farZ) 函数来初始化的。投影矩阵还可以基于一个现存的平截体形状计算出来。

11.5　小结

本章探讨了 3D 图形的数学基础。限于篇幅很难涵盖线性代数的深度和广度。

矩阵是通过存储用于定义三个坐标轴方向的单位矢量，以及坐标系原点在参考坐标系中的位置来编码坐标系的。所有的坐标系都是相对于其他坐标系的。默认 OpenGL 坐标系提供了一个基础参考，其他坐标系是基于这个基础参考来定义的。

矩阵还定义了视平截体的形状、位置和方向。可以在矩阵和平截体数据结构间来回转换。四元法可以编码关于任意矢量的旋转，四元法可以用于避免旋转一个矩阵出现错误时的未定义行为。矩阵和四元数之间也是可以来回转换的。

第 12 章会介绍一个结合了地形渲染、天空盒、粒子系统、动画、变化视点、灯光、模型和碰撞检测技术的例子。要描述出在一个应用中应该在何时，以及为什么而使用前面章节中分隔介绍的专题可能有些困难。第 12 章会把所有的知识都结合在一起。

PDG

第 12 章　理清整体思路

本章会介绍一个简单的模拟程序（例子 OpenGLES_Ch12_1）的设计和构造。这个模拟程序结合了地形、特效、模型、动画、视点变化、纹理、灯光和优化。本章会构建一个适度复杂的整合了几乎前面章节中介绍的所有代码的应用。

在例子 OpenGLES_Ch12_1 中，你将会控制在图 12-1 中显示的三个火箭推进的悬空车之一。你可以通过倾斜 iOS 设备来控制方向，通过按下 Boost 按钮来加速。这些车可以相互碰撞，并可以被撞出航线。这个模拟会通过重力、空气阻力和一些简单的人工智能（AI）来控制剩下的两个车。

图 12-1　模拟程序的屏幕截图

12.1　概述

例子 OpenGLES_Ch12_1 是基于第 10 章中的例子扩展来的，具体是通过向 iOS 标准 Model-View-Controller（MVC）应用体系结构的 Model 和 View 子系统中添加

类来实现的。新的 TECart 和 TEParticleEmitter 类会扩展 Model 子系统以呈现悬空车和封装发射粒子的 3D 位置。在 View 子系统中，OpenGLES_Ch12_1 会添加一个新的 UtilityBillboardParticleEffect 类以及 OpenGL ES 2.0 Shading Language 程序来渲染由公告牌组成的粒子。公告牌特效会渲染朝向观察者的纹理几何对象，如第 8 章所述。

模型视图控制器

模型视图控制器应用体系结构是最悠久、最成功的复用软件设计模式之一。这种模式是 20 世纪 70 年代首先在 Smalltalk 编程语言中提出的。模型视图控制器模式定义了 Cocoa Touch 框架的整体组织结构。模型视图控制器模式是把大量的协作对象组织为不同的子系统的一个高级模式，子系统包括：模型、视图和控制器。

对于常见应用的功能和行为的分析有助于理解这些子系统在模型视图控制器模式中所起的作用。大部分应用会存储信息、恢复信息、向用户呈现信息，以及让用户编辑或者操作信息。在面向对象的应用中，信息并不只是字节；对象封装了信息以及使用信息的方法。一个复杂应用中的每个对象都应该属于下述子系统中的一个。

- 模型。模型子系统包含用于实现一个应用的独特功能和信息存储的对象。不要混淆了 3D 图形中的模型概念与模型子系统中的模型概念。MVC 的模型子系统会包含处理应用数据的所有规则。最重要的是模型子系统可以单独存在，不用依赖于视图或者控制器子系统。在例子 OpenGLES_Ch12_1 中，地形高程数据是存储在模型子系统中的。高程数据以及执行在这些数据上的操作可以独立于形象化地形的很多可能的方式而存在，其中地形是指从平面地形图到交互性纹理 3D 网格的地形。
- 视图。视图子系统会呈现从模型子系统收集来的信息，并会为用户提供一个与信息进行交互的途径。可以存在呈现相同模型的多个视图子系统。例如，可能会有一个 3D 视图、一个 2D 视图、一个由表格组成的视图、一个打印报表视图、一个命令行视图、一个基于 Web 的视图和一个脚本语言视图与同一个模型进行交互。在第 7 章和第 10 章介绍的 COLLADAViewer 和 TerrainEditor 应用演示了数据的多个视图，包括 3D 渲染和图表。
- 控制器。控制器子系统会让模型与视图脱钩。用户与视图子系统的交互会产生对于控制器子系统的请求，控制器可能转而改变在模型子系统中的信息。控制器还操作着呈现给用户的数据的转换和格式化。例如，一个模型子系统可能是以米来存储数据的，但是根据用户偏好，控制器子系统可能会转换数据的单位为尺或者像素。一个模型子系统可能会以无序的集合来存储对象，但是在向视图子系统提供这些对象以呈现给用户之前，控制器可能会排序这些对象。

MVC 确保了在修改模型子系统的实现时不会影响视图子系统，反之亦然。在例子 OpenGLES_Ch12_1 中，用户会按下“Boost”按钮，这个按钮是视图子系统的一部分。

这个按钮会向一个 OpenGLES_Ch12_1ViewController 实例发送一个 Objective-C 消息“-startBoosting：”，这个实例接着会向模型子系统内的玩家控制的 TECart 实例发送一个消息。作为响应，TECart 实例会重新计算加速矢量，这个加速矢量用于在物理模拟中为火箭车计算出一个新的速度和位置。下次视图子系统渲染场景时，玩家的车会由于这个加速度而出现在一个新的 3D 位置。

在 MVC 体系结构中，火箭车会应用自己的物理规则来计算加速度、速度和位置，不会考虑开始加速的原因和火箭车将会被可视化呈现的方式。在 3D 场景内的一个位置对于火箭车的渲染与用来确定火箭车的位置所做的运算是无关的。对于火箭车物理效果实现代码的改变不会影响火箭车的渲染代码，对于渲染代码的改变不会影响火箭车物理效果的实现代码。图 12-2 描绘了在例子 OpenGLES_Ch12_1 中的 MVC 关系。图中的箭头代表了子系统间的通信。

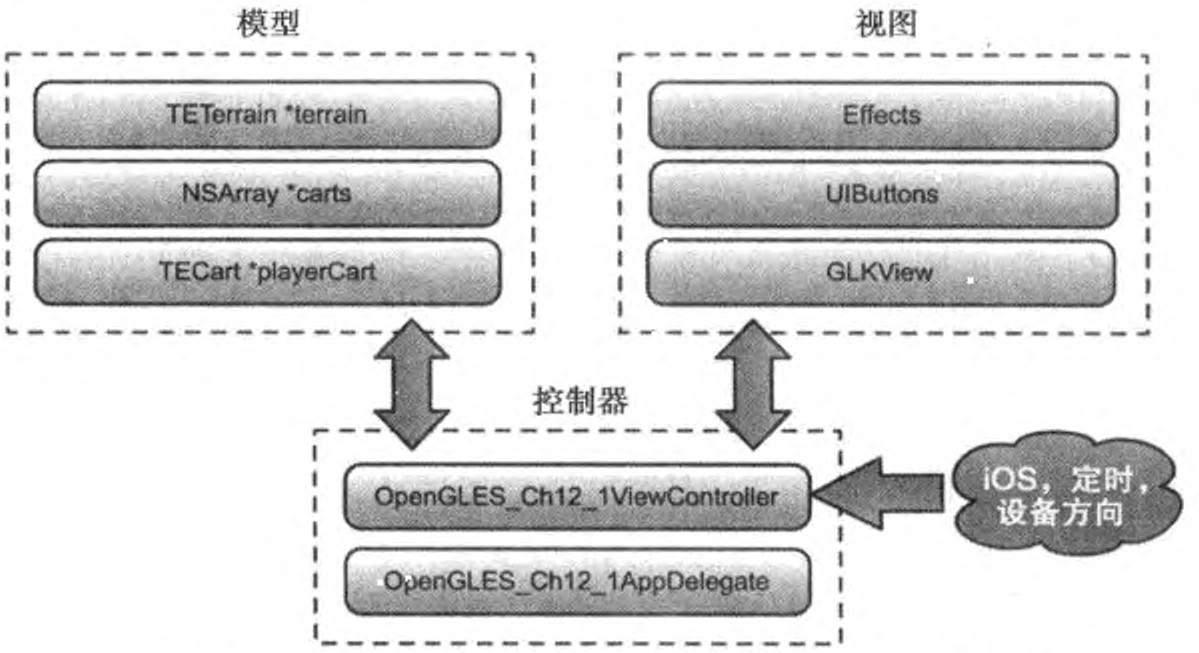


图 12-2　例子 OpenGLES_Ch12_1 中的子系统之间的通信

Cocoa Touch 通过引入模型控制器和视图控制器的概念改进了 MVC 结构。在例子 OpenGLES_Ch12_1 中，OpenGLES_Ch12_1AppDelegate 类起到了模型控制器的作用。它是有责任加载和保存类似地形和火箭车的模型子系统对象的控制器子系统的一部分。OpenGLES_Ch12_1ViewController 类起到了视图控制器的作用，用于协调模型子系统和视图子系统间的通信。OpenGLES_Ch12_1ViewController 用于管理时间、命令模型子系统对象更新内部状态，以及接收来自 GLKView 实例的消息以协调渲染。

12.2　一切如故

本书前面的例子完全是在视图和控制器子系统中演示 3D 概念的。这些简单的例子要么完全缺少一个模型子系统，要么有一个不完整的模型子系统，比如封装在视图控制

器中的一个三角形或者一个简单的网格。随着应用复杂性的增加，把代码分割成不同的 MVC 子系统的迫切性也会增加。几十年的使用证明了 MVC 结构对于组织复杂应用的价值。类似 UIView、GLKView、UIViewController 和 GLKViewController 的 Cocoa Touch 类的名字强调了它们的作用。当把一个全功能的 iOS 应用所需的所有部件汇集在一起时，几乎总是使用 MVC 结构才能够实现最好的代码组织形式。

接下来，简单地介绍了组成例子 OpenGLES_Ch12_1 的类。这里描述的作用、关系和交互会阐明每种技术在你自己的应用中可能会使用的方式。

12.2.1　控制器子系统

例子 OpenGLES_Ch12_1 中的控制器子系统仅仅由两个类组成，这两个类的实现都会尽可能地使用适量的代码，两个类一共大约由 1200 行代码组成。

1. OpenGLES_Ch12_1AppDelegate

与第 10 章的例子类似，例子 OpenGLES_Ch12_1 会使用苹果的 Core Data 框架来加载地形数据和 3D 模型。Core Data 的存在是为了简化模型子系统内的复杂的对象和关系的存储。OpenGLES_Ch12_1AppDelegate 类包含了与 Core Data 进行交互所需的所有代码。

类 OpenGLES_Ch12_1AppDelegate 还会使用在第 7 章介绍，并在第 9 章继续探讨的方法来从一个 .modelplist 文件中加载一个火箭推进悬空车的 3D 模型。第 7 章讲解了紧凑型二进制“.modelplist”文件格式以及它的优点。类 OpenGLES_Ch12_1AppDelegate 会让应用的其余部分与模型对象的存储细节分隔开来。不同的 OpenGLES_Ch12_1AppDelegate 实现会使用完全不同的技术来加载地形和模型数据，同时不会对例子 OpenGLES_Ch12_1 的其余部分产生影响。

2. OpenGLES_Ch12_1ViewController

类 OpenGLES_Ch12_1ViewController 管理着模型和视图子系统对象的周期性更新。天空盒实现了完整的地形效果所需的广阔远景。粒子为渲染场景添加了交互性，有助于表现火箭车悬空于地面的效果。天空盒和粒子特效是在第 8 章讲解的。

类 OpenGLES_Ch12_1ViewController 还会跟踪显示刷新率，并会在刷新率明显下降时，试着适时地减少绘图细节，就像在旧式 iOS 设备或者 iPad 和 iPhone 模拟器上所做的一样。在每帧中渲染复杂的地形网格和数以千计的粒子会加重 GPU 的负担。类 OpenGLES_Ch12_1ViewController 会降低绘制简化的地形网格时的阀值距离，并会在刷新率低于 20Hz 时中断粒子的渲染。根据分析以及苹果公司的 OpenGL ES 性能检测应用，当运行在 iPad 2 上时，例子 OpenGLES_Ch12_1 并不会受限于 GPU 的性能。

OpenGLES_Ch12_1ViewController 会把方法整理成组，这些组包括：“Load time configuration”、“Device orientation”、“View lifecycle”、“Accessors”、“GLKViewDelegate

updating”、“GLKViewDelegate drawing”、“Camera delegate”和“Responding to gestures and actions”。这些组涵盖了 GLKViewController 子类所起的所有作用，除了 Camera delegate。对于所有的方法来说，哪一个方法要归于哪一类中并没有特定的原因，但是典型的基于 GLKit 的应用大概会包含所有的这些方法组。

12.2.2 模型子系统

地形、火箭车和物理效果的模拟是在模型子系统内实现的。在 Xcode 工程 OpenGLES_Ch12_1 内的一个 Core Data“Entity Relationship”（实体关系）文档（OpenGLES_Ch12_1.xcdatamodeld）中记录了地形和模型位置的信息。Core Data 实现了地形和模型位置的存储，以便在例子 OpenGLES_Ch12_1 启动时可以很容易地加载信息。这个 Xcode 工程还包含两个由 Xcode 根据 Core Data 文档生成的类，TETerrain 和 TEModelPlacement。通常，没有必要为了封装 Core Data“Entities”（实体）而生成类。在例子 OpenGLES_Ch12_1 中这样做主要是为了实现 Objective-C 类别的创建，通过创建类别来添加与地形和模型进行交互的方法。

生成的 TETerrain 和 TEModelPlacement 类形成了模型子系统的核心，但是本节中介绍的其他几个类也是需要的。例如，TECart 类封装了悬空车的相关信息。TECart 实例是在控制器子系统中由 OpenGLES_Ch12_1AppDelegate 按需创建的，这个实例会履行模型控制器的主要责任，即加载或者创建模型子系统对象。其余的模型子系统类会被用于 TETerrain、TEModelPlacement 和 TECart 类的实现中。例如，TETerrain 会使用 UtilityTextureInfo 来存储地形纹理数据，使用 UtilityMesh 和 UtilityModel 来存储 3D 模型数据。TECart 会使用在 AGLKCollision 和 AGLKFilters 中声明的函数来实现物理效果。

1. AGLKCollision

在 AGLKCollision.h 和 AGLKCollision.m 文件中包含着非常有用的用于检测 3D 对象之间的碰撞的函数。尤其是下面的 AGLKRayDoesIntersectTriangle() 函数会计算一条光线与一个三角形之间的交点。回想一下，地形网格是由数以千计的三角形组成的。给定一个在地形的 X 和 Z 位置的对象，以及一条从这个对象上直接投射下来的光线，通过 AGLKRayDoesIntersectTriangle() 函数引用返回的 intersectionPoint 提供了这条光线与一个地形三角形表面的交点。AGLKRayDoesIntersectTriangle() 函数用于让火箭车看起来悬空在地形表面之上。

```
extern BOOL AGLKRayDoesIntersectTriangle(
   GLKVector3 rayDirection,
   GLKVector3 pointOnRay,
   GLKVector3 trianglePointA,
   GLKVector3 trianglePointB,
   GLKVector3 trianglePointC,
   GLKVector3 *intersectionPoint)
```

如果在这条光线与三角形之间找到一个交点，那么 AGLKRayDoesIntersectTriangle() 函数会返回 YES，如果找不到交点就返回 NO。结果 NO 表明一个对象不在地形之上；也就是说这个对象驶出了地形的边界。

光线是由在这条光线上的任意一点 pointOnRay 以及用来指定这条光线的方向的一个矢量 rayDirection 组成的。火箭车的位置和 Y 轴的负方向共同定义了一条从火箭车投射出来的光线。任意三点，trianglePointA、trianglePointB 和 trianglePointC 定义一个三角形。由三个角组成的地形网格三角形用于计算光线与地形的交点。

2. AGLKFilters

在 AGLKFilters.h 和 AGLKFilters.m 文件中包含着用于低通滤波的函数，低通滤波是在第 6 章讲解的。在例子 OpenGLES_Ch12_1 的多个地方使用了低通滤波函数，以实现流畅的动画效果所需的增量变化。例如，当一个悬空车转向时，逐渐转向要比直接跳转到一个新方向的效果更好。逐渐转向是使用 AGLKVector3LowPassFilter() 过滤函数实现的，具体会增量更新代表火箭车的当前朝向的矢量，直到与新目标方向一致。

```
extern GLKVector3 AGLKVector3LowPassFilter(
   GLfloat fraction,
   GLKVector3 target,          // target value to approach
   GLKVector3 current);        // current value
```

目标矢量指定了一个新的值。当前矢量指定了当前值。参数 fraction 用于指定这个函数返回的矢量与目标矢量或者当前矢量之间的接近程度。例如，如果 fraction 等于 0.0，那么返回值与当前值相同。如果 fraction 等于 1.0，那么返回的值与目标值相同。当 fraction 的值介于 0.0 与 1.0 之间时，返回的矢量会对应于目标矢量跟当前矢量之间的一个矢量。

动画效果是随着时间变化值而产生的，具体是通过使用一个逐渐变化的 fraction 参数调用一个过滤函数来实现的。fraction 参数通常是通过使用自动画开始起的逝去时间，除以完成这个动画所需的总时间计算出来的。例如，一个火箭车只有在从开始转向起所逝去的时间的总数量，等于或者超过为转向所分配的时间量时，才会完成转向。

3. AGLKFrustum

第 5 章介绍了平截体的数学概念，角锥体形状描述了被坐标系变换置为可见的空间范围。第 9 章介绍了 AGLKFrustum 数据类型和利用它执行计算的函数。AGLKFrustum 存储了编码在标准的投影和 model-view 变换矩阵中的相同信息。AGLKFrustum.h 文件中声明了用来初始化 AGLKFrustum 数据类型的、在 AGLKFrustum 和矩阵之间来回转换的，以及决定点或者球是否在平截体内的函数。在模型子系统中声明的 AGLKFrustum 类型是一个纯粹的数学结构，无论特定视图子系统的实现是怎么样的，这个结构都是有效的。在 UtilityCamera 类中和模型子系统的模拟中都使用了 AGLKFrustum 数据类型。

4. TETerrain（modelAdditions）

modelAdditions 类别向 Core Data 生成的类 TETerrain 添加了方法。添加的这些方法简化了地形相关的计算。例如，与在 Core Data 中用来存储值的单位不同，“-widthMeters”方法会返回以米为单位的整个地形的宽度。“-calculatedHeightAtXPosMeters:zPosMeters:surfaceNormal:”方法会使用在 AGLKCollision.h 中声明的函数来返回在地形的 X 和 Z 位置处的高度。这个方法还会通过引用返回一个指定了在碰撞点处的地形的坡度矢量。坡度在物理模拟中具有重要作用，有助于确定重力对于火箭车的影响力。

5. TECart

每个 TECart 实例都有对应的一个 3D 模型、一个 3D 位置和一个速度，并会存储任意“推进”加速的当前量。还有多个其他的属性是基于位置和速度按需计算出来的，比如用于定义火箭车前进方向的一个矢量。火箭车会向它的可选委托对象报告自身位置的改变，以给予它的委托一个影响这个改变的机会。例如，在控制器子系统中的 OpenGLES_Ch12_1AppDelegate 实例充当了火箭车的委托，它会限制火箭车的位置以防止火箭车跑出地形的边缘。在 Cocoa Touch 应用中，使用控制器对象作为模型和视图对象的委托是非常普遍的。

TECart 实例实现了在“- (void) updateWithController:(id <TECartControllerProtocol>) controller”方法中的物理模拟。下面在 TECart.h 文件中声明的 TECartControllerProtocol 协议指定了控制器必须要实现的方法。

```
//////////////////////////////////////////////////////////////////////////
// Protocol for objects that control carts.
@protocol TECartControllerProtocol

- (NSTimeInterval)timeSinceLastUpdate;
- (TETerrain *)terrain;
- (UtilityBillboardParticleManager *)particleManager;
- (TECart *)playerCart;

@end
```

TECartControllerProtocol 协议的“-timeSinceLastUpdate”和“-terrain”方法可用于计算火箭车在地形表面上的位置。“-particleManager”方法没有在模型子系统内使用，但是当 TECart 的“-(void)emitParticlesWithController:(id <TECartControllerProtocol>) controller”方法被视图子系统内的 TECart（viewAdditions）类别覆写后，这个方法就可以用于实现粒子效果。TECart 使用“-playerCart”方法来决定火箭车是受玩家控制还是受人工智能控制。

6. TEParticleEmitter

TEParticleEmitter 类存储了一个位置属性并声明了一个用于控制粒子发生器的 Objective-C 块类型。对于火箭车来说，粒子发生器的位置是相对于火箭车的位置定义的。在火箭车的 3D 模型中包含的一个小的三角金字塔形的 3D 模型指定了粒子发生器的位置。在类似谷歌 SketchUp 的 3D 模型编辑工具中，一般使用一个小的 3D 模型来显示粒子发生器的位置。

TEParticleEmitter 实例是在“-(id)initWithModel:(UtilityModel *)aModel”方法中初始化的。aModel 参数的几何中心指定了粒子发生器的位置。

实际的粒子生成器位于类似如下类型的块中：

Actual generation of particles happens within a Block with the following type:

```
//////////////////////////////////////////////////////////////////
// Type for blocks called to emit particles
typedef void (^TEParticleEmitterBlock)(
   GLKVector3 position,
   UtilityBillboardParticleManager *manager,
   NSTimeInterval elapsedTime,
   id owner);
```

通过向 TEParticleEmitter 的“-updateWithParticleEmitterBlock:manager:elapsedTime: owner:”方法传递块，能够灵活地实现多个粒子效果。使用块可以在不改变 TEParticle-Emitter 类本身的情况下变化粒子发生逻辑。

7. UtilityBillboardParticle

每个 UtilityBillboardParticle 实例都会存储 position、velocity、force、initialSize、finalSize、lifeSpanSeconds、fadeDurationSeconds, minTextureCoords 和 maxTextureCoords 属性。这些属性都可以用来产生动画：改变位置会移动粒子。通过 velocity 和 force 属性可以实现基于逝去时间和物理学来计算新位置的方式。粒子会从初始大小通过逐渐变换而长大或者缩小为最终大小。当一个粒子走完它的生命周期后，就不必再绘制这个粒子了。粒子透明度会根据 faceDurationSeconds 的值在完全不透明（1.0）和完全透明（0.0）之间变化。如果 faceDurationSeconds 等于 0.0，那么这个粒子永远不会褪色。如果 fadeDurationSeconds 大于 lifeSpanSeconds，那么这个粒子在它第一次被绘制时是半透明的。

UtilityBillboardParticle 还会按需计算 distanceSquared、isAlive、lifeRemainingSeconds、size 和 opacity 属性，以提供粒子的当前状态。当这个粒子被通过“-updateWith-ElapsedTime:frustum:”方法而更新时，以及当这个粒子被用于根据视平截体眼睛位置解平方而来的有符号距离而排序粒子的计算时，就需要计算 distanceSquared 属性。

粒子渲染是在视图子系统内实现由 UtilityBillboardParticleManager 类执行的。视图子系统需要排序粒子以产生正确的渲染结果，具体内容将在“UtilityBillboard Particle

Effect 和 Utility Billboard Particle Shader”一节中讲解。

虽然现在还没有实现，但有效的公告牌属性可以用于计算公告牌与模型子系统内的其他对象之间的碰撞。例如，在一个射击游戏中，基于公告牌的粒子可以被用于表现子弹或者激光束。公告牌和其他对象之间的碰撞可以被模型子系统内的物理模拟器侦测到。

8. UtilityTextureInfo

UtilityTextureInfo 类封装了存储在二进制“.modelplist”文件中的原始纹理数据。在模型子系统内，UtilityTextureInfo 只提供了简单的存储。这个类被视图子系统内的 UtilityTextureInfo(viewAdditions) 类别扩展而实现了 UtilityTextureInfo 实例和 GLKTextureInfo 实例间的转换，以在 GLKit 和 OpenGL ES 中使用。

9. UtilityMesh

UtilityMesh 类首先是在第 6 章介绍的，用于管理大型数量的几何顶点数据，以及命令处理数据的子范围。网格是 3D 图形中要用到的关键数据结构。网格定义了 3D 模型的形状。虽然地形数据是使用 TETerrain 类而不是 UtilityMesh 类来表示的，但地形使用的仍然是网格概念的一个变体。在 AGLKCollision.h 中声明地用来检测 3D 对象之间的碰撞的函数，也能够用于检测任意三角形组（包括网格）之间的碰撞。

用于渲染网格的方法是由在视图子系统中实现的 UtilityMesh(viewAdditions) 类别提供的。通常在模型子系统中实现抽象的、几何学的、数学的方法以及数据存储，然后在视图子系统内扩展对于这些数据的操作。无论渲染怎么执行，这些几何运算都是有效的，并且支持用于类似碰撞检测的非图形计算。使用在视图子系统中实现的类别可以从通用几何中分离出特殊的渲染方法。例如，组成一个网格的几何数据可以被显示为一个表中的坐标值。使用表的视图子系统可以用一个类别来扩展 UtilityMesh 类，这个类别可以包含为 Cocoa Touch UITableView 对象提供数据的方法，而不是用于 3D 渲染的方法。

10. UtilityModel

每个 UtilityModel 实例会存储一个 mesh 属性，这个 mesh 属性用于定义一个或者多个 3D 模型的几何图形。UtilityModel 还会存储一个 name 属性，用于指定 3D 模型所需的特定网格子集。计算出来的 axisAlignedBoundingBox 属性提供了一个边界框，这个边界框容纳了一个模型所包含的所有网格顶点。属性 doesRequireLighting 指定了是否要逐个使用 OpenGL ES 灯光计算来渲染 3D 模型。

第 6 章介绍了 UtilityModel 类以及它与 UtilityMesh 类之间的关系。网格对大型数量顶点数据的存储做了优化，具体是使用一个适合向 GPU 控制的内存转移的格式来存储。在一次操作中向 GPU 发送所有的网格数据就是一个性能优化。然后 3D 模型会协调已经存储在 GPU 控制的内存中的网格顶点子集的渲染。在视图子系统内的 UtilityModel（viewAdditions）类别实现了对于 UtilityModel 实例的渲染。

12.2.3　视图子系统

视图子系统包含了用于显示地形、模型，以及类似按钮的用户界面对象的代码。很多在模型子系统中声明的类是使用在视图子系统内的类别来扩展出特殊的渲染能力的。为了理解模型和视图子系统之间的去耦合和分离，必须要问一个关键的问题："是否能够基于相同的模型子系统来实现完全不同的视图子系统，是否能够在不影响视图子系统的情况下改变模型子系统的实现？"例子 OpenGLES_Ch12_1 证明这些是可以办到的。

例如，模型子系统可以在不影响视图子系统的情况下，使用一个数据存储结构来代替 Core Data。视图子系统可以在不影响模型子系统的情况下，被完全地从工程中省略掉；在例子 OpenGLES_Ch12_1 中的模型子系统不含有对于视图子系统的引用或者依赖。

在本节讲解的类和类别演示了怎么为去耦合性的模型子系统所提供的抽象几何图形数据实现相对复杂并且全功能的 3D 渲染。视图子系统中的代码仍然适用于很多由纹理 3D 模型、公告牌和粒子组成的 3D 应用。

1. UtilityEffect

第 10 章介绍了用于加载、编译、连接和验证 OpenGL ES 2.0 Shading Language 程序的 UtilityEffect 类。UtilityEffect 遵守 GLKit 的 GLKNamedEffect 协议。UtilityEffect 会实现两个方法，"-bindAttribLoactions" 和 "-configureUniformLocations"，当这两个方法被直接调用时会生成异常。UtilityEffect 的子类必须要实现这两个方法。

UtilityEffect 的 "-(BOOL)loadShadersWithName:(NSString *)aShaderName" 方法会加载并编译顶点和片元 Shading Language 程序，使用由 aShaderName 参数指定的根名，以及后缀 / 扩展名 ".vsh" 和 ".fsh"。"-loadShadersWithName:" 方法会按需调用 "-bindAttribLocations" 和 "–configureUniformLocations" 函数，如果碰到任何问题就返回 NO，如果成功就返回 YES。

在例子 OpenGLES_Ch12_1 中的 UtilityModelEffect、UtilityPickTerrainEffect、UtilityTerrainEffect 和 UtilityBillboardParticleEffect 类都是 UtilityEffect 的子类。它们实现了 "-bindAttribLocations" 和 "-ConfigureUniformLocations" 方法，以绑定属性并配置特定于它们各自的 Shading Language 程序的统一值存储。

2. UtilityModelEffect 和 UtilityModelShader

UtilityModelEffect 类会协调 UtilityModelShader.vsh 顶点 Shading Language 程序和 UtilityModelShader.fsh 片元 Shading Language 程序的执行。UtilityModelShader 程序是相对简单的，并且只使用一个纹理、一个环境光和一个定向漫反射光来渲染提交的几何图形数据。因为简单所以可以使用传统 OpenGL 灯光计算的一个子集来有效地渲染 3D 模型。第 3 章讲解了纹理。第 4 章讲解了环境和漫反射光的灯光方程。

UtilityModelShader.fsh Shading Language 程序会丢弃所有接近透明的片元。丢弃它

们可以防止对于深度渲染缓存的无效更改，当片元重叠时，深度渲染缓存会决定新生成的片元是否要替换先前生成的片元。例如，假如没有丢弃，当为一个模型生成的一个接近透明的片元，处于为另一个模型生成的一个不透明片元的前面时，那么接近透明的片元会阻止不透明的片元对最终像素颜色渲染缓存内容的影响，从而导致一个错误的渲染结果。在这种情况下，正确的结果应该是能够透过接近透明的片元看到不透明的片元，但是又不能完全显示不透明的片元。第 5 章介绍了深度渲染缓存，第 3 章讲解了用来影响片元颜色的操作。把接近透明的片元丢弃掉可以避免问题，并且不会降低场景的渲染质量，因为接近透明的片元是很难察觉到的。

3. UtilityPickTerrainEffect 和 UtilityPickTerrainShader

第 10 章讲解了 UtilityPickTerrainEffect 类以及相关的用来渲染伪色彩地形和模型的 UtilityPickTerrainShader Shading Language 程序。场景会被渲染入一个由 UtilityPickTerrainEffect 实例控制的像素颜色渲染缓存中。地形的 X 和 Z 位置坐标会被编码入片元的红色和绿色颜色分量中。例如，由地形的 {100,50,75} 位置生成的片元颜色是浮点 RGB 颜色 {100.0 / terrainWidth, 75.0 / terrainLength, 0.0}。在像素颜色渲染缓存的特定位置的结果像素颜色指定了对应于这个位置的地形坐标。3D 模型索引是与编码在像素颜色渲染缓存内的像素的蓝色分量中的伪色彩一起渲染的。当从像素颜色渲染缓存中读回一个像素颜色时，如果这个颜色有一个非零的蓝色分量，那么这个蓝色分量的值就是用来查询这个像素颜色所对应模型的一个索引。

在例子 OpenGLES_Ch12_1 中并没有使用 UtilityPickTerrainEffect 类，但是保留了第 10 章所实现的 UtilityPickTerrainEffect 类。在 OpenGLES_Ch12_1 中利用拾取的代码就留给读者来完成吧。例如，触摸事件可以被用于实现从玩家的火箭车向触摸位置发射导弹。如果玩家能够通过估计当导弹到达时火箭车的所在位置来消灭计算机控制的火箭车的话，这个模拟程序会变得更像一个游戏。玩家可能还要躲避计算机控制的火箭车发来的导弹。

4. UtilityTerrainEffect 和 UtilityTerrainShader

第 10 章讲解了 UtilityTerrainEffect 类以及相关的用来渲染纹理地形的 UtilityTerrainShader Shading Language 程序。这个 Shading Language 程序会把 4 个纹理混合在一起来计算地形几何体所产生的片元的颜色。每个纹理会被施加不同的纹理变换矩阵和混合权重，这样做可以实现多种类型的最终渲染结果。第 3 章讲解了纹理变换。用来混合纹理的权重会与预计算的灯光强度值一起存储在第 5 个纹理中。换句话说，灯光被烘焙进了第 5 个纹理中，因此这个 UtilityTerrainShader Shading Language 程序就没有必要执行灯光计算了。

地形渲染会应用一些在本书中用过的最复杂的 Shading Language 程序，但是即便这样，在第一代 iOS 设备上，仍然可以有很好的性能表现。如果使用早前不支持 Shading Language 程序的 OpenGL ES 版本的话，想要获得同等的性能表现和渲染结果几乎是不可能的。

5. UtilityBillboardParticleEffect 和 UtilityBillboardParticleShader

UtilityBillboardParticleEffect 类以及相关的 UtilityBillboardParticleShader Shading Language 程序会渲染半透明粒子。粒子需要一个稍微不寻常的顶点属性。

```
//////////////////////////////////////////////////////////////////
// Vertex attributes used in UtilityBillboardParticleShader
typedef struct
{
  GLKVector3 position;
  GLKVector2 textureCoords;
  GLfloat opacity;
}
BillboardVertex;
```

每个顶点的 position 和纹理坐标 textureCoords 属性比较简单，并且与类似 UtilityModel-Effect 的其他效果中使用的顶点属性相似。但是，公告牌粒子顶点存储了一个 opacity 属性，而没有使用顶点法向量属性或者顶点颜色属性。透明度用在 Shading Language 程序中，用于控制公告牌片元与像素颜色渲染缓存中的现存颜色之间的混合。这个 opacity 属性相当于红绿蓝和透明度（RGBA）颜色中的透明度分量。这种只存储顶点的透明度而不是全部颜色的方法，既可以节省内存的消耗，又可以提高运行时的性能。

因为粒子是半透明的，所以它们需要在其他的类似地形、模型和天空盒的不透明对象被渲染后再渲染。这样做可以让在像素颜色渲染缓存中的最终颜色描绘透过公告牌粒子观察对象的画面。这些粒子也必须要基于跟观察者的距离按照从远到近的顺序来绘制。否则，就不可能透过近处的公告牌粒子看到远处的公告牌粒子。图 12-3 显示了期望的渲染结果。对于公告牌粒子的顶点属性以及按照与观察者距离排序的粒子的管理是在 UtilityBillbordParticleManager 类中实现的。

图 12-3　渲染半透明公告牌

6. UtilityBillboardParticleManager

每个 UtilityBillboardParticleManager 类都会维持 UtilityBillboardParticle 实例以及与之对应的顶点数据的数组。每次一个 UtilityBillboardParticleManager 准备好绘制时，它都会基于每个粒子与平截体眼睛位置之间的距离排序 UtilityBillboardParticle 实例的数组。

```
//////////////////////////////////////////////////////////////////
// Sort billboards in reverse order for rendering from furthest
// to nearest.
- (void)prepareToDrawWithCamera:(UtilityCamera *)aCamera;
```

UtilityBillboardParticleManager 会收集包含在 aCamera 的平截体内的存活粒子的顶点属性，然后以一个单独的大型三角形集合来提交用于渲染的顶点属性。存活粒子是 isAlive 属性为 YES 的粒子。在公告牌经过了 lifeSpanSeconds 所指定的时间后，UtilityBillboardParticle 的 isAlive 属性会变成 NO。

“-addParticle:”方法会向 UtilityBillboardParticleManager 实例所维持的公告牌数组中添加一个粒子。

```
//////////////////////////////////////////////////////////////////
// Adds aParticle to the end of the particles array. If a
// dead particle is available, then one dead particle is also
// removed so the total number of particles does not increase.
- (void)addParticle:(UtilityBillboardParticle *)aParticle;
```

为了简化创建新的 UtilityBillboardParticle 实例并把它添加到管理器过程中的步骤，UtilityBillboardParticleManager 提供了如下能够一次完成这个过程的方法。

```
- (void)addParticleAtPosition:(GLKVector3)aPosition
   velocity:(GLKVector3)aVelocity
   force:(GLKVector3)aForce
   initialSize:(GLKVector2)anInitialSize
   finalSize:(GLKVector2)aFinalSize
   lifeSpanSeconds:(NSTimeInterval)aSpan
   fadeDurationSeconds:(NSTimeInterval)aDuration
   minTextureCoords:(GLKVector2)minCoords
   maxTextureCoords:(GLKVector2)maxCoords;
```

7. UtilityModelManager

UtilityModelManager 类封装了一个网格、一个纹理，以及一个能够使用部分的网格和纹理来渲染的 3D 模型的集合。第 7 章介绍了 UtilityModelManager 类并讲解了可以用于从二进制数据中加载网格和 3D 模型的方法。3D 模型是由 UtilityModelManager、UtilityModel、UtilityMesh、UtilityModelEffect, 以 及 UtilityModelShader Shading Language 程序共同处理的，从以二进制数据的形式加载它们开始，一直到像素颜色渲染缓存内的最终渲染结果。

UtilityModelManager 充当了 3D 模型和网格之间的总管理者或者总协调者。对于 UtilityModelManager 的“-prepareToDraw”方法的调用还会通过用网格顶点数据配置当前 OpenGL ES 上下文的方法来准备要绘制的网格。方法“-modelNamed:”会返回请求的模型，返回的模型可以接着通过准备一个用于绘图的 UtilityModelEffect 来渲染，然后调用 3D 模型的“-draw”方法。

8. UtilityCamera

第 10 章介绍了 UtilityCamera 类，以封装一个视平截体并提供用来移动和旋转虚拟 3D 世界中的摄像机的便利方法。每当摄像机的位置发生变化时，摄像机也会通知一个可选的委托对象。这个委托有对摄像机的变化做出影响的机会。第 10 章中的例子使用了委托来保持摄像机在地形的上面。例子 OpenGLES_Ch12_1 没有明确地限制摄像机的位置，因为这个例子会一直让摄像机在玩家所控制的火箭车的附近，而火箭车被它们自己的委托限制在了地形的上面。

9. UtilityMesh (viewAdditions)

类别 UtilityMesh (viewAdditions) 扩展了模型子系统内的 UtilityMesh 类。下面是这个类别所添加的方法。

```
@interface UtilityMesh (viewAdditions)
- (void)prepareToDraw;
- (void)prepareToPick;
- (void)drawCommandsInRange:(NSRange)aRange;
- (void)drawBoundingBoxStringForCommandsInRange:
   (NSRange)aRange;
```

“-prepareToDraw”方法会使用网格顶点属性来设置当前 OpenGL ES 上下文，以用于接下来的渲染。“-prepareToPick”方法与“-prepareToDraw”方法类似，但是当使用 UtilityPickTerrainEffect 渲染时，“-prepareToPick”方法会设置当前 OpenGL ES 上下文去使用一个顶点属性的子集。“-drawCommandsInRange:”方法会调用 glDrawElements() 函数一次或者多次以渲染由网格顶点属性所定义的三角形。参数 aRange 通常会指定用来标识一个 3D 模型所使用的顶点属性的顶点索引的一个子集，但是理论上可以一次绘制多个 3D 模型。最后，“-drawBoundingBoxStringForCommandsInRange:”方法有时可以帮助可视化调试 3D 模型的渲染。这个方法会沿着一个盒子的边绘线，这个盒子围住了指定范围的命令所引用的顶点。就像“-drawCommandsInRange:”方法，这个指定的范围通常只包含一个 3D 模型，因此最终的盒子会形象地围住这个模型。

10. UtilityModel (viewAdditions)

类别 UtilityModel (viewAdditions) 会向模型子系统内的 UtilityModel 类添加一个额

外的方法。这个添加的方法会调用关联网格的“-drawCommandsInRange:”方法，这个方法指定了适合模型的命令范围。

```
@implementation UtilityModel (viewAdditions)

//////////////////////////////////////////////////////////////////
// This method draws the receiver using the receiver's
// UtilityMesh and a UtilityModelEffect that have both already
// been prepared for drawing.

- (void)draw
{
   [self.mesh drawCommandsInRange:NSMakeRange(
      indexOfFirstCommand_, numberOfCommands_)];
}


@end
```

11. TETerrain (viewAdditions) 和 TETerrainTile

地形网格由于太大而无法在一次 glDrawElements() 函数调用中绘制完成，绘制整个网格的话会连看不到的部分也绘制了，这明显是一种浪费。因此，地形网格会被分割为封装在 TETerrainTile 类中的瓦片。每个瓦片会引用组成封装在 TETerrain 实例中的整个地形的地形网格顶点的一个小型子集。第 10 章介绍过 TETerrainTile 类、TETerrain 类、以及 TETerrain (viewAdditions) 类别。

除了地形网格本身，TETerrain 类的实例会引用 TEModelPlacement 实例。每个模型位置指定了地形内的一个 3D 模型的名字、位置和方向。例子 OpenGLES_Ch12_1 会使用模型位置来绘制树木、灌木、岩石和建筑物。没有理由绘制位于看不到的瓦片中的 3D 模型。由于这个原因，TETerrain (viewAdditions) 类别提供了一个只绘制在指定的瓦片内的模型的方法。

```
- (void)drawModelsWithinTiles:(NSArray *)tiles
   withCamera:(UtilityCamera *)aCamera
   modelEffect:(UtilityModelEffect *)aModelEffect
   modelManager:(UtilityModelManager *)modelManager;
```

传入“-drawModelsWithinTiles: withCamera:modelEffect:modelManager:”方法的 UtilityModelManager 对象会使用指定的 UtilityModelEffect 来协调 3D 模型的渲染，利用指定 UtilityCamera 提供的变换矩阵。相似的方法会自己绘制瓦片，使用一个特定的摄像机和 UtilityTerrainEffect 类。

12. TECart (viewAdditions)

类别 TECart (viewAdditions) 向模型子系统内的 TECart 类添加了“-(void)drawWithEffect:

(UtilityModelEffect *)anEffect”方法。

```
////////////////////////////////////////////////////////////////////
// Draw the receiver using anEffect.
- (void)drawWithEffect:(UtilityModelEffect *)anEffect;
{
   GLKVector3 position = self.position;

   // Move cart to position
   anEffect.modelviewMatrix = GLKMatrix4Translate(
      anEffect.modelviewMatrix,
      position.x,
      position.y,
      position.z);

   // Transform to cart's orientation
   anEffect.modelviewMatrix = GLKMatrix4Multiply(
      anEffect.modelviewMatrix,
      GLKMatrix4Transpose(self.orientationMatrix));

   [anEffect prepareModelview];

   [self.model draw];
}
```

火箭车的方向是在模型子系统内计算的，用于让火箭车悬空在地形上，即使地形是倾斜的，火箭车也会按需倾斜。火箭车的方向是相对于火箭车自身的，是由它的 orientationMatrix 属性定义的，但这个矩阵并不是渲染一个火箭车模型时用来修改 UtilityModelEffect 的变换矩阵的。相反，在 UtilityModelEffect 的自身坐标系中必须要使用一个相应的方向矩阵。为了把火箭车的自身坐标系转换成 UtilityModelEffect 的坐标系，就有必要逆转火箭车的 orientationMatrix。“-drawWithEffect:”方法就是利用这一事实来实现均匀的坐标系缩放的，计算代价较小的 GLKMatrix4Transpose() 函数与较复杂的 GLKMatrix4Invert() 函数是等效的。使用转置函数而不用逆函数的这个小优化带来了一个不明显的限制，即火箭车模型不应该被不均匀地缩放。不要试图渲染一个压扁的火箭车。

TECart (viewAdditions) 类别的实现覆写了“-(void) emitParticlesWithController:(id <TECartControllerProtocol>)controller”方法，这个方法是在模型子系统的 TECart 总实现中定义的。在视图子系统内对这个方法的新的实现会使用火箭车的 particleEmitter 属性和 UtilityBillboardParticleManager 来创建粒子模拟火箭排气管、飞尘和烟雾，参见图 12-4。

13. UtilityTextureInfo (viewAdditions)

UtilityTextureInfo 类封装了一个属性字典，包括用于定义纹理图片的二进制数据属性，但是这个属性不适合直接在 OpenGL ES 中使用。UtilityTextureInfo (viewAdditions)

类别提供了 name 和 target 属性，用于 OpenGL ES 确定接下来的渲染所需的一个纹理。第 3 章讲解了纹理的名字和目标。为了生成所需的名字和目标，而在幕后创建了一个新的 GLKTextureInfo 实例，并用 UtilityTextureInfo 的图像数据初始化了这个实例。然后就可以从 GLKTextureInfo 实例中获得名字和目标。

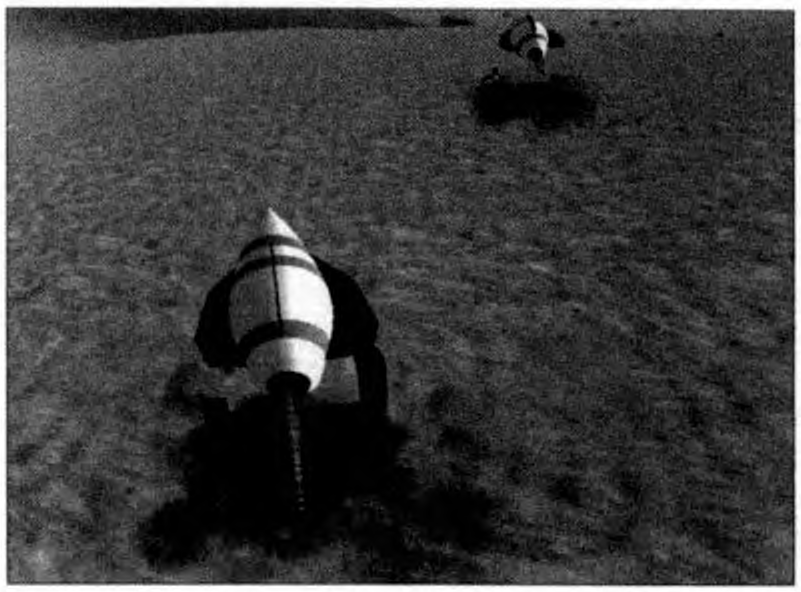


图 12-4　火箭车上的粒子发生器

为了从 UtilityTextureInfo 实例创建 GLKTextureInfo 实例而用下面的类别扩展了 GLKTextureInfo 实例。向框架类添加方法的能力是 Objective-C 编程语言最强大的功能之一。

```
@implementation GLKTextureInfo (utilityAdditions)

//////////////////////////////////////////////////////////////////
// Returns a GLKTextureInfo instance initialized based upon
// information provided in aDictionary.
+ (GLKTextureInfo *)textureInfoFromUtilityPlistRepresentation:
   (NSDictionary *)aDictionary
{
   GLKTextureInfo *result = nil;

   const size_t imageWidth = (size_t){[aDictionary
      objectForKey:@"width"] unsignedIntegerValue];
   const size_t imageHeight = (size_t)[[aDictionary
      objectForKey:@"height"] unsignedIntegerValue];

   // The imageData property is expected to be a Tiff image
   UIImage *image = [UIImage imageWithData:
      [aDictionary objectForKey:@"imageData"]];

   if(nil != image && 0 != imageWidth && 0 != imageHeight)
   {  // Create GLKTextureInfo corresponding to image.
      NSError *error;
      result =
         [GLKTextureLoader textureWithCGImage:[image CGImage]
```

```
            options:[NSDictionary dictionaryWithObjectsAndKeys:
               [NSNumber numberWithBool:YES],
               GLKTextureLoaderGenerateMipmaps,
               [NSNumber numberWithBool:NO],
               GLKTextureLoaderOriginBottomLeft,
               [NSNumber numberWithBool:NO],
               GLKTextureLoaderApplyPremultiplication,
               nil]
            error:&error];

   if(nil == result)
   {
      NSLog(@"%@", error);
   }
   else
   {
      glTexParameteri(GL_TEXTURE_2D,
         GL_TEXTURE_MAG_FILTER,
         GL_LINEAR);
      glTexParameteri(GL_TEXTURE_2D,
         GL_TEXTURE_MIN_FILTER,
         GL_NEAREST_MIPMAP_LINEAR);
      glTexParameteri(
         GL_TEXTURE_2D,
         GL_TEXTURE_WRAP_S,
         GL_REPEAT);
      glTexParameteri(
         GL_TEXTURE_2D,
         GL_TEXTURE_WRAP_T,
         GL_REPEAT);
      }
   }

   return result;
}

@end
```

12.3　设备动作

Cocoa Touch 包含了 Core Motion 框架，这个框架提供了 CMMotionManager 类，详情请访问：https://developer.apple.com/library/ios/DOCUMENTATION/CoreMotion/ Reference/CoreMotion_Reference/_index.html。例子 OpenGLES_Ch12_1 中的 OpenGLES_Ch12_1ViewController 类在“-viewDidLoad:”方法中使用下面的代码初始化了它的 motionManager 属性并启动了周期性的动作更新。

```
self.motionManager = [[CMMotionManager alloc] init];
[self.motionManager startDeviceMotionUpdates];
```

OpenGLES_Ch12_1ViewController 实现了“-viewDidUnload”方法以终止动作更新并把 motionManager 属性设置为 nil。

```
[self.motionManager stopDeviceMotionUpdates];
[self.motionManager stopAccelerometerUpdates];
self.motionManager = nil;
self.particleManager = nil;
```

较旧的 iOS 设备的硬件并不能检测到所有的动作。当比较精确的动作检测器不可用时，OpenGLES_Ch12_1ViewController 的“-updatePlayerCartForDeviceMotion”方法会退而使用内置但是不太准确的加速计硬件。

```
//////////////////////////////////////////////////////////////////
// Factor used to dampen motion inputs controlling cart yaw
static const GLfloat TEDeviceRotationFactor = (0.03f);

//////////////////////////////////////////////////////////////////
// This method updates the player's cart orientation in response
// to device motion. Turning the device turns the player's cart.
- (void)updatePlayerCartForDeviceMotion
{
   TECart *playerCart = [self playerCart];

   // Update cart direction based on device motion inputs
   if(self.motionManager.isDeviceMotionActive)
   {
      self.rollAngleRadians = TEDeviceRotationFactor *
         self.motionManager.deviceMotion.attitude.pitch;
   }
   else
   {  // Use device accelerometer when other sources unavailable
      if(!self.motionManager.isAccelerometerActive)
      {
         [self.motionManager startAccelerometerUpdates];
      }

      self.rollAngleRadians = -TEDeviceRotationFactor *
         self.motionManager.accelerometerData.acceleration.y;
   }

   [playerCart turnDeltaRadians:self.rollAngleRadians];
}
```

图 12-5 显示了用来控制玩家的火箭车的设备动作。CMMotionManager 把设备绕着指出屏幕的坐标轴的旋转定为了“俯仰”。加速计把指出设备屏幕的坐标轴定为了

“Y”。因此，如果可能的话，“-updatePlayerCartForDeviceMotion”会获取设备 attitude 的俯仰变化，否则会获取绕着设备 Y 轴的旋转加速度。

术语 attitude 描述了一个对象就俯仰、滚转和偏航三个角度而言的方向。如果一个对象有一个定为“前”的方向，那么滚转是指绕着指向前的轴的旋转。偏航是指绕着指向上的轴的旋转。俯仰是指绕着与前和上都垂直的方向的轴的旋转。维基百科提供了一个图解，以及一些使用飞机来演示俯仰、滚转和偏航的动画，详情请访问：http://en.wikipedia.org/wiki/Yaw,_pitch_and_roll。

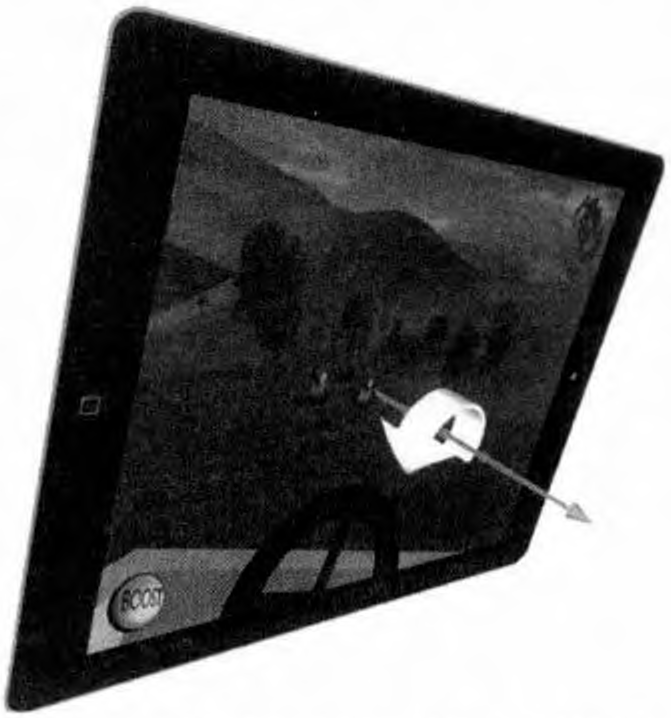


图 12-5　用来控制火箭车的设备旋转图示

12.4　小结

本章挨个介绍了计算机图形的概念，最后用一个比较综合的例子包含了几乎所有涉及的主题。苹果公司在 Cocoa Touch 框架中使用了 MVC 软件结构，大部分 iOS 应用也应该采用 MVC 结构。其他熟悉 Cocoa Touch 的程序员所期望的一个组织形式是把类和其他的源代码划分成模型、视图和控制器子系统。例子 OpenGLES_Ch12_1 就是受益于 MVC 组织形式所提倡的去耦化。

很多经验丰富的图形程序员年复一年积累了很多可重用代码库。无须从基本原理开始一步一步地创建一个新工程的每个高级功能。尽管如此，对于所涉及技术的理解仍然是很重要的。从类似 GLKit 的高级框架开始启动一个工程有很多好处，会让你成为一个多产的程序员，但可能也会缺少对于理论和实践的理解。理解了底层技术，那么类似 GLKit 和第三方游戏引擎的高级框架使用起来也更容易。通常，程序员是最讨厌神秘的一类人。知道所有东西是怎么结合到一起的是非常重要的。

最后，iOS 设备屏幕上的每个像素颜色都是由 GPU 控制的，而 GPU 是使用标准的 OpenGL ES 2.0 编程接口控制的。苹果公司的 Core Animation 框架和 GLKit 简化了很多常见的图形编程任务，并且不会影响对于底层 OpenGL ES 2.0 函数的访问。本书讲解了怎么使用 GLKit 和 Core Animation 层来渲染 3D 图形，但是更重要的是，本书提供了对于 GLKit 实现的深入理解，以及与 Core Animation 的相互作用。使用这些知识去实现你自己的创意吧！