OpenGL ES 2.0 编程向导

Aaftab Munshi
Dan Ginsburg
Dave Shrener

翻译： 江湖游侠 QQ：86864472 email：mazhaoyang2005@gmail.com
本翻译花费作者 3 个月有余，能力有限，错误在所难免，欢迎交流、指出错误，转载请注明出处，

装载着色器
创造一个项目目标并连接着色器
设置视口并清楚颜色缓冲区
装载几何图形并画一个
在后缓冲区显示物体

3. 一个 EGL 指导例子
   如何窗口系统联系
   检查错误
   EGL 初始化
   可用的配置属性
   EGL 配置属性
   EGL 选择属性
   创建一个 On-Screen 区域：EGL 窗口
   创建一个 Off-Screen 区域：EGL 缓冲
   创建一个贴图
   做一个 EGL 贴图
   连接我们的 EGL 在一起
   同步
4. 着色器和编程
   着色器和编程
   创建着色器
   创建链接一个项目
   表格和属性
   得到和设置
   得到和设置属性
5. OpenGL ES 着色器语言
   OpenGL ES 着色器语言基础
   变量和变量类型
   构造变量
   矢量和矩阵成员
   常量
   结构
   数组
   操作器
   函数
   内置函数
   控制语句
   表格
   属性
   变量
   预处理程序和指令
   表格和变量 pack
   精度和限定
   恒定

6. 顶点属性、顶点矩阵和缓冲区目标
   指定顶点属性数据
      顶点属性常量
      顶点矩阵
   在顶点着色器里声明顶点属性
      绑定顶点属性
      在顶点着色器了改变
   顶点缓冲区目标
   缓冲区目标匹配
      Tip 性能
7. 基本图元装配和光栅化
   基本图元
      三角形
      直线
      点类型
   基本 draw
      执行 tips
   集合
      坐标系统
      边
      透视分割
      视口转换
   旋转
      选择
      多边形设置
8. 顶点着色器
   顶点着色器预览
      顶点着色器变量
      向前兼容
      ES 2.0 顶点着色器限制
   顶点着色器例子
      一个顶点着色器例子
      顶点着色器光照
   普通贴图坐标系
   顶点皮肤
   OpenGL ES 1.1 固定顶点管道与 ES 2.0 顶点着色器
9. 贴图
   贴图基础
      2D 贴图
      立方体贴图
      贴图对象和装载贴图
      贴图过滤和 Mip 贴图
      自动 Mip 贴图产生
      贴图坐标系封装

1 什么是 OpenGL ES?

OpenGL ES 是一个为便携或嵌入式设备例如：移动电话、PDAs、监视器、等，发展的 3D 绘图 APIs（应用编程接口）。OpenGL ES 是一个被叫做 Khronos 联盟的组织创造的一系列 APIs 中的一个。Khronos 联盟成立于 2000 年一月，是一个专注为手持和嵌入式设备建立开放标准和自由版权的 APIs 的工业联盟。

在桌面设备中有两个 3D API 接口，DirectX 和 OpenGL。DirectX 是运行 Window 系统平台事实上的 3D 标准，在 3D 游戏平台中主要的标准。OpenGl 是一个跨平台的 3D API 标准，能够运行在 linux 系统，各种 UNIX 系统，Mac OS X 和 Wicrosoft 系统等等。它是广泛接受的 3D API，这个 API 被 Doom 和 Quake 系列游戏使用。用于 Mac OS X 的接口，在 CAD 应用像 CATIA、数字建模应用像 Maya 和 SoftImage|XSI。

因为 OpenGL 作为 3D API 的广泛使用，这是很自然的从桌面的 OpenGL 开始发展为手持设备和便携式开放标准的 3D API; 又因为使用 OpenGL ES 的设备地址空间和内存的限制、低的内存带宽、敏感的电源功耗、缺乏浮点运算硬件，所以修改它以适用与手持和便携设备领域。工作组对 OpenGL ES 做了下面的一些定义限制。

OpenGL ES API 是巨大的和复杂的，工作组的目标是建立一个适合设备驱动程序。为此工作组删除了一些 OpenGL API 的冗余设计，对同一种操作有多种实现方法，只留下主要的，多余的技术被删除，例如：一个几何体，OpenGL 中可以使用立即模式、显示列表、顶点矩阵，在 OpenGL ES 中只能使用顶点矩阵，显示列表和立即模式被移除。

移除冗余是重要的，但保持兼容性也是重要的。尽可能的 OpenGL ES 被设计成 OpenGL 的嵌入式子集，OpenGL 将也能运行 OpenGL ES 的程序。原因是作为一个子集能更好的应用桌面程序的代码和工具。虽然如此，也有偏离，特别是 OpenGL ES 2.0，细节将在后面的章节讨论。

手持和便携设备的限制被引入，例如减少电力消耗、减少着色器性能，渲染质量被引入着色器语言。

OpenGL ES 的设计者目标是确保最小的图像质量，大多数的手持设备只用一个有限的屏幕，在屏幕上做图尽可能的保持好的质量，将是重要的一项工作。

QpenGL ES 工作组提供确保各种 OpenGL ES 设备正常工作、提供好的图像质量、健壮性的测试，这些设备必须通过完整的测试。

一共有三个OpenGL ES白皮书被释放，OpenGL ES 1.0、OpenGL ES 1.1和OpenGL ES 2.0 OpenGL ES 1.0 和 OpenGL ES 1.1 描述固定功能管线从 OpenGL1.3 和 OpenGL1.5 发展而来，OpenGL ES 2.0 描述可编程绘图管线，发展自 OpenGL 2.0,因为发展自 OpenGL，意味着 OpenGL 是 OpenGL ES 的基础，决定着未来 OenGL ES 的版本，

OpenGL ES 2.0 来自 OpenGL 2.0 API,确保了对游戏的支持，例如在颜色调色板 1 用的图片来自 demo 球游戏，OpenGL ES 2.0 的目标和他使用的着色器优良的效果例如环境绘制和 per-fragment 光照。这个游戏的例子在 OpenGL ES 2.0 是很普遍的。有了 OpenGL ES 2.0 在桌面的应用将可以用于便携设备。

在本章里，我们给 OpenGL ES2.0 的管线一个介绍。

OpenGL ES 2.0

OpenGL ES 2.0 是本书中要讲的 API，目标是讲述完整的细节（核心和扩展），给出怎么使用的例子，讨论各种优化技术，读完本书希望能你能懂的 OpenGL ES 2.0 的核心 API,能够开发 OpenGL ES 2.0 的应用程序，不要担心技术细节，懂得怎么去工作。

OpenGL ES 2.0 包含两部分：OpenGL ES 2.0 API 说明和 OpenGL ES 着色器语言说明，图 1_1 显示 OpenGL ES 2.0 图像管线，图 1 _1 中的的着色器盒子描述了 OpenGL ES 2.0 的管道可编程阶段，OpenGL ES 2.0 可编程管线的每个阶段预览在下面介绍。

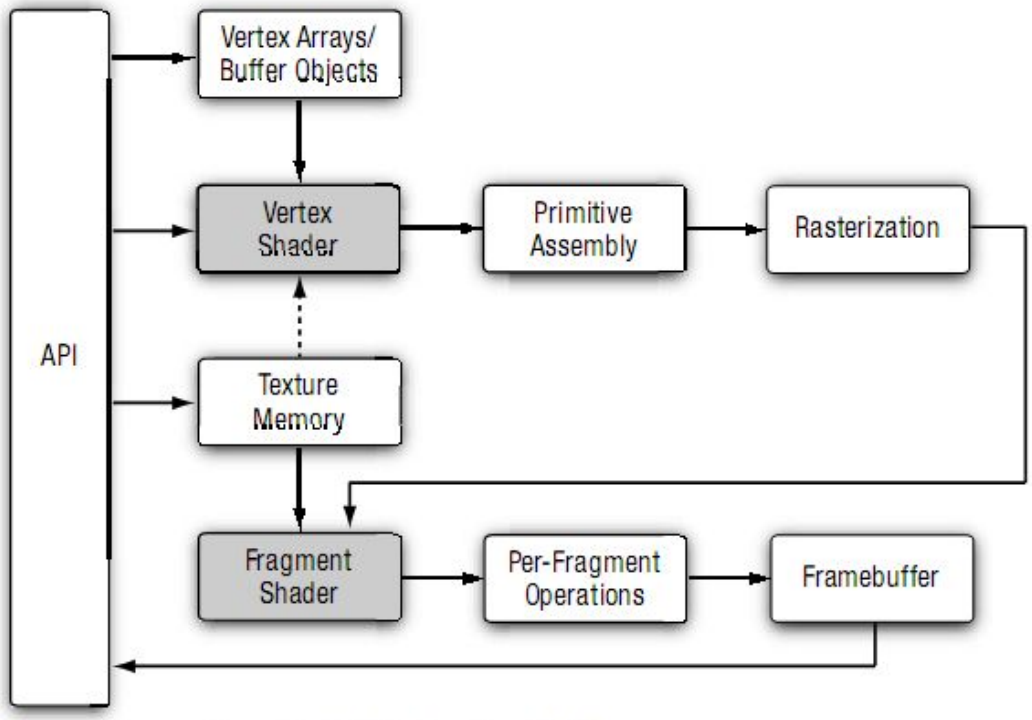


Figure 1-1 OpenGL ES 2.0 Graphics Pipeline

顶点着色器

本节给出概述，后面章节深入介绍。

顶点着色器一般的编程方法;

属性—顶点矩阵支持的 Per_vertex 数据

Uniforms—顶点着色器使用的常量数据

Samplers—被 Uniforms 使用的特殊类型，在顶点着色器的贴图中使用，是可选项

着色器编程—顶点着色器编程源码或可执行的部分

顶点着色器的输出叫做 varying 变量，在最初的光栅化阶段，这些变量被计算，作为片段着色器的输入，从顶点着色器的矩阵使用插补的方法产生片段着色器的变量，输入和输出在图 1_2 描述:

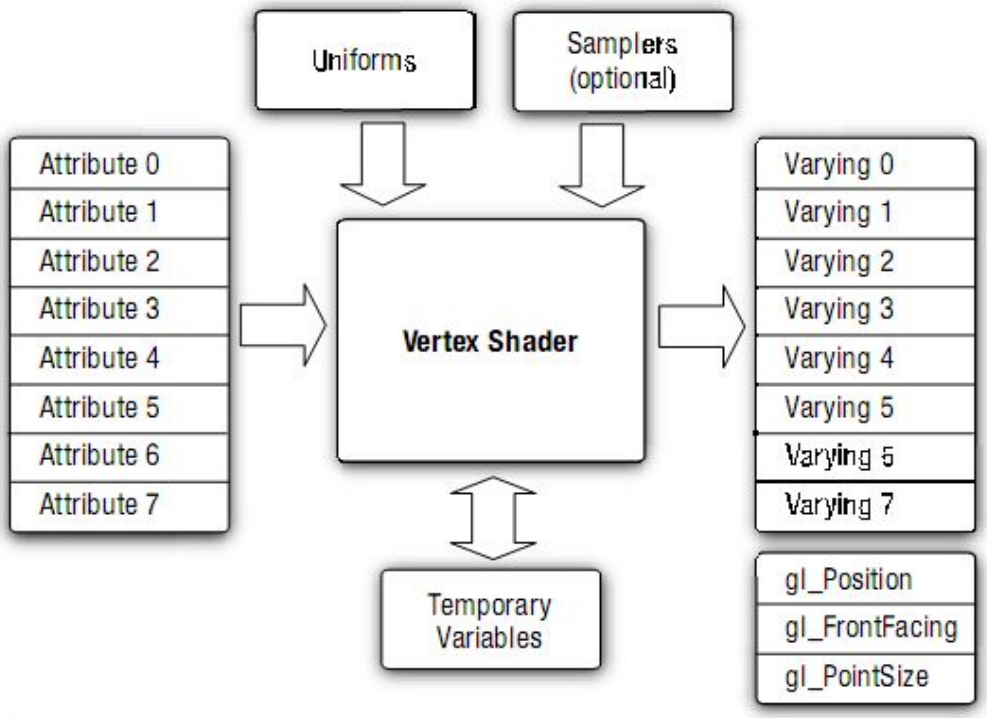


Figure 1-2 OpenGL ES 2.0 Vertex Shader

顶点着色器被使用在传统的基于顶点的操作，例如位移矩阵、计算光照方程、产生贴图坐标。顶点着色器被应用指定，应用于客户的顶点转化。

例 1_1 显示使用 OpenGL ES 编程语言编写顶点着色器，后面章节再介绍编程的细节，现在仅需要知道它的大致就可以了，顶点着色器需要一个位置和颜色数据作为输入属性，输入位置数据是 4×4 的矩阵，输出是变换后的位置和颜色。

Example1-1 A Vertex Shader Example

```
1.  // uniforms used by the vertex shader
2.  uniform mat4     u_mvpMatrix; // matrix to convert P from model
3.                                // space to normalized device space.
4.
5.  // attributes input to the vertex shader
6.  attribute vec4   a_position; // position value
7.  attribute vec4   a_color;    // input vertex color
8.
9.  // varying variables C input to the fragment shade
10. varying vec4     v_color;    // output vertex color
11.
12. void
13. main()
14. {
15.     v_color = a_color;
16.     gl_Position = u_mvpMatrix * a_position;
17. }
```

第 2 行定义一个 uniform 变量 u_mvpMatrix 应用存储模型的关联视图和投射矩阵，6、7 行定义输入顶点矩阵和顶点属性，10 行定义顶点着色器输出的变量 per_vertex。编译时 gl_Position 被编译器自动定义，片段着色器的唯一的入口点是 main 函数，15 行读入顶点属性 a_color，输出顶点颜色 v_color，16 行变换后的矩阵位置输出到 gl_Position。

基元装配

基元是能够被 OpenGL ES 绘制的几何物体，这些绘图命令描述了一个顶点属性和基元几何体以及基元类型的集合。顶点属性包括计算位置、颜色和贴图坐标的信息，他们将被输入到片段着色器中。

顶点着色器能绘制的几何图元包括三角形、直线、点，对每个图元必须判断是否位于投影平截体内，如果图元不完全在平截体内部，将被视图平截体剪贴，如果完全在平截体外，将被丢弃，然后顶点位置被转变为屏幕坐标，剔除操作也能够舍弃一些图元，依据图元位于正面还是背面，剪切和剔除后，图元进入光栅化阶段。

光栅化

图 1_3 所示，光栅化是转化图元为二维片段的过程，被片段着色器执行，二维的片段像素能够被绘制在屏幕上。

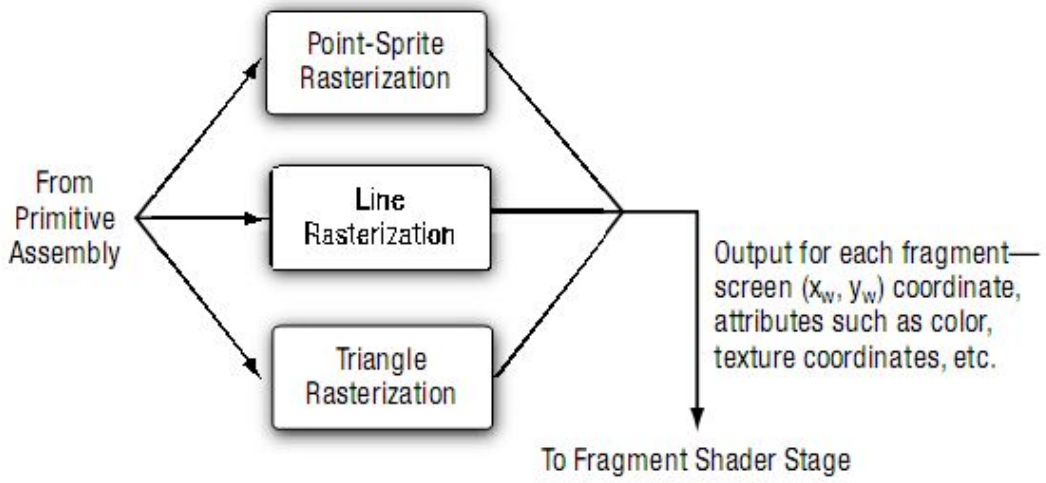


Figure 1-3 OpenGL ES 2.0 Rasterization Stage

片段着色器

如图 1_4 片段着色器使用下面的输入，执行光栅化后形成片段。

Varying 变量—光栅化阶段使用插补技术为片段着色器产生，顶点着色器的输出

Uniforms—常量数据

Samplers—Uniforms 中特殊的类型，在贴图时使用

着色器程序—片段着色器编程源码或可执行的部分

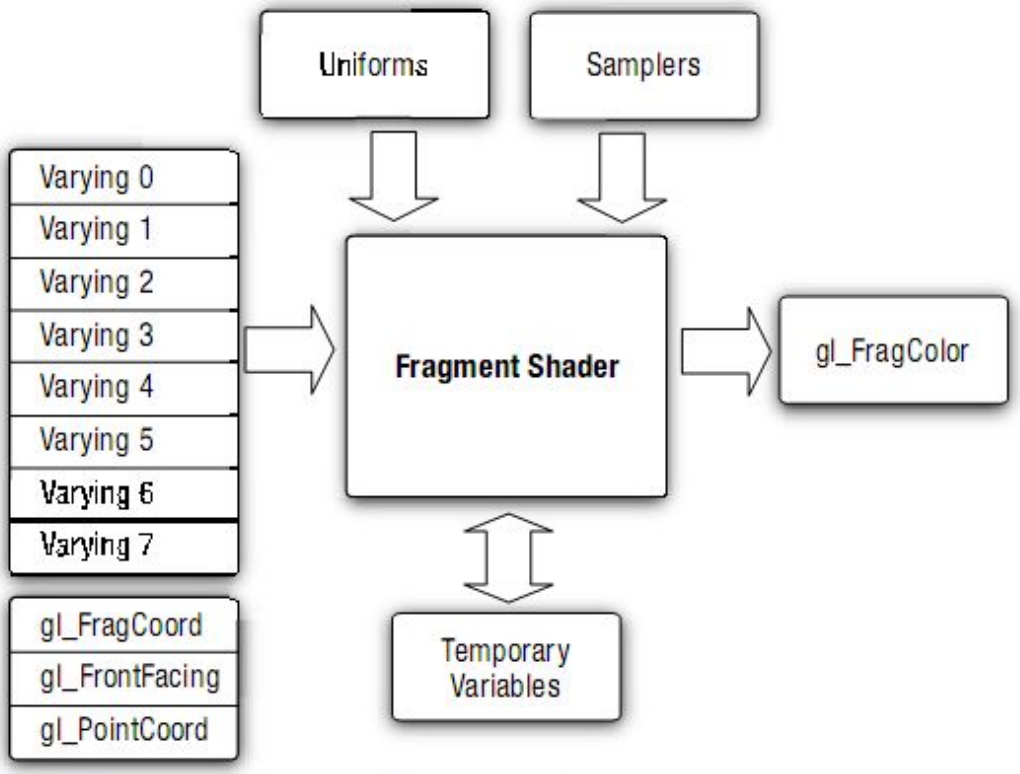


Figure 1-4 OpenGL ES 2.0 Fragment Shader

片段着色器也能丢弃片段或者产生一些颜色值像 gl_FragColor，光栅化阶段产生的颜色、深度、模板、屏幕坐标（Xw，Yw）变成 OpenGL ES 2.0 管线输入，例 1_2 显示一个简单的片段着色器例子，它绘制一个 gouraud 三角形。

Example1-2 A Fragment Shader Example

```
1. precision mediump float;
2.
3. varying vec4      v_color;    // input vertex color from vertex shader
4.
5.
6. void
7. main(void)
8. {
9.         gl_FragColor = v_color;
10.}
```

第 1 行设定显示质量，第三行描述片段着色器输入，6-10 行描述主函数，片段着色器不需定义输出，这是因为片段着色器仅仅的输出是 gl_FragColor，第 9 行设定片段着色器的输入是 v_color。

Per-fragment 操作

下一个操作是 per-fragment，per-fragment 操作执行下面的操作。

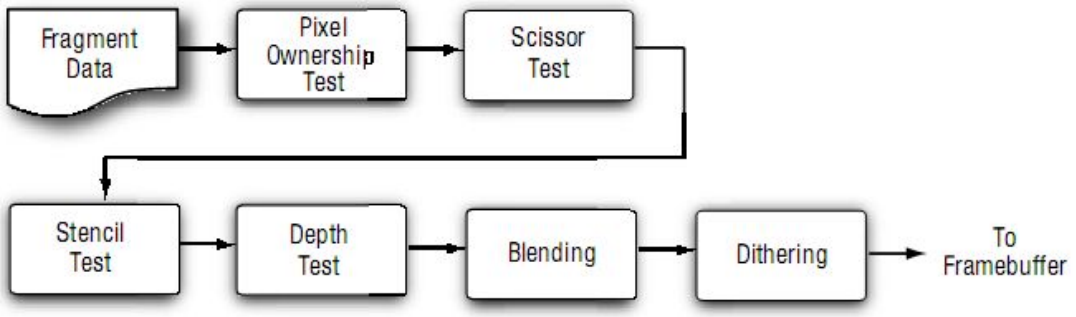


Figure 1-5 OpenGL ES 2.0 Per-Fragment Operations

像素所有权测试——这个测试决定在帧缓冲区中某点（Xw,Yw）的像素当前是否被 OpenGL ES 所有，这个测试准许视窗操作系统去控制帧缓冲区中的像素是否是 OpenGL ES 的内容，例如视窗检测到 OpenGL ES 的帧缓冲区被另一个窗口遮挡，视窗系统可以决定遮挡 OpenGL ES 的内容，不显示。

剪切测试—测试（Xw,Yw）是否在剪切矩形内，如果在矩形外，片段被丢弃。

模板深度测试—决定输入的片段是否应该被拒绝

混合—混合新产生的片段颜色和存储在颜色缓冲区中的颜色

抖动—被使用在用几种颜色的组合模拟出大范围内的多种色彩模式

per-fragment 阶段最后，片段颜色、深度值、模板值是否被写入帧缓冲区还要依靠各自的掩码位是否被使能，例如，颜色缓冲区能设定红色是否能被写入颜色缓冲区。

最后 OpenGL ES 2.0 也支持缓冲区数据读回功能，但仅仅颜色缓冲区像素能够读回，深度和模板缓冲区数据不能读回。

注意：透明度测试和 LogicOp 测试不再是 per-fragment 测试的部分，这两部分在 OpenGL 2.0 和 OpenGL ES 1.x 存在，透明度测试不再需要，因为片段着色器能丢弃片段，透明度测试能在片段着色器阶段执行。另外 LogicOp 被移除，因为它很少被使用，OpenGL ES 工作组不再要求厂商（ISVs）支持这个特征。

OpenGL ES 2.0 和 OpenGL ES 1.x 向后兼容性

OpenGL ES 2.0 不向后兼容 OpenGL ES 1.x，它不支持固定功能的管线。OpenGL ES 2.0 可编程顶点着色器取代 OpenGL ES 1.x 固定功能顶点着色器的应用，OpenGL ES 2.0 工作组决定不提供向后兼容能力主要基于如下考虑：

提供对固定功能管线的支持暗示 API 将支持多种的方法执行一个特性，这违背了设计的初衷，可编程管线即使提供做固定管线的支持，也不会获得更多的注意。

对硬件供应商来说，那将有更多的工作要做，而可编程管线能提供更多的灵活性。

即支持固定管线，支持可编程着色器意味着设备需要更多的内存，分离固定管线和可编程管线，厂商也不需要更多的驱动支持。

而且，不像 OpenGL ES 1.x，OpenGL ES 2.0 那也没有配置文件或扩展命令。

EGL

OpenGL ES 要求一个渲染上下文环境和绘图窗口，渲染上下文存储着 OpenGL ES 状态的属性，绘图窗口决定那个窗口将被绘制，绘图窗口指出将要绘制的图形对颜色缓冲区、深度缓冲区、模板缓冲区的对要求，还有每个缓冲区位数。

OpenGL ES API 没有提及如何创造一个渲染上下文或者渲染上下文如何联系操作系统的窗口，EGL 是 Khronos 组织创造的渲染 API（OpenGL ES）和操作系统窗口之间的接口。当提供 OpenGL ES 时，不是必需的提供 EGL，开发者可以根据平台厂商公布的文档决定使

用何种接口。

任何 OpenGL ES 在使用 EGL 渲染前需要做下面的事情：

查询设备上可用的显示初始化它，例如一个翻盖手机有两个 LCD 屏幕，我们可以渲染其中的一个和两个。

创造一个渲染屏幕，EGL 能够创造隐藏平面或显示屏平面，显示屏平面联系着操作系统，隐藏平面联系着像素缓冲区，虽然不用于显示，但被使用于贴图渲染，能被多种 Khronos API 使用。

创造渲染上下文，EGL 需要创造渲染上下文，这个渲染上下文在实际使用前需要联系上合适的平面。

EGL API 也有另外的功能，例如电力管理，支持多种渲染上下文，分享贴图或者缓冲区等等，通过 OpenGL ES 扩展行为能够通过工具查询获得。

最新的 EGL 版本是 1.4.

OpenGL ES 2.0 编程

编程开始前你需要知道那些包含头文件和你的应用链接所用的库文件位置，懂得使用 EGL 语法和 GL 命令名和命令参数。

库和包含文件

应用程序需要的链接库包括 OpenGL ES 2.0 库 libGLESv2.lib 和 EGL 库 libEGL.lib。

需要包含的头文件至少有：

```
#include <EGL/egl.h>
#include <GLES2/gl2.h>
#include <GLES2/gl2ext.h>
```

Egl.h 是 EGL 头文件，gl2.h 是 OpenGL ES 2.0 头文件，gl2ext.h 是 OpenGLES 批准的扩展。

头文件和库文件命名与平台有关，OpenGL ES 2.0 工作组试图定义库文件和头文件名，但它们可能没包括所有的平台，所以开发者可以参照供应商的文档说明，确定库文件和头文件的命名和组织。

EGL 命名语法

所有的 EGL 函数以 egl 前缀开始（例如 eglCreateWindowSurface），EGL 数据类型开始 EGL 前缀例如 EGLint 和 EGLenum。

表 1_1 给了一个 EGL 数据类型的简明描述。

Table1-1 EGL Data Types

Data Type	C-Language Type	EGL Type
32-bit integer	int	EGLint
32-bit unsigned integer	unsigned int	EGLenum EGLBoolean
32-bit pointer	void *	EGLConfig，EGLContext, EGLDisplay, EGLSurface, EGLClientBuffer

OpenGL 命令语法

所有的 EGL 命令以前缀 gl 开始，组成命令名的每个单词的首个字母大写，（例如 glBlendEquation），同样 OpenGL ES 数据类型也以前缀 GL 开始。

另外，一些命令能够接受不同类型的参数，数据类型（byte [b], unsigned byte [ub],short [s], unsigned short [us], int [i], fixed [x], and float [f]）可能包括有 1 到 4 种，部分函数也接受矢量类型。

下面两个命令是相同的，只是输入参数一个是浮点数，另一个是整型数。

```
glUniform2f(location, 1.0f, 0.0f);
```

glUniform2i(location, 1, 0)

下面的命令也是相同的，只是输入参数一个是矢量，另一个不是。

```
GLfloat     coord[4] = { 1.0f, 0.75f, 0.25f, 0.0f };
glUniform4fv(location, coord);
glUniform4f(location, coord[0], coord[1], coord[2], coord[3]);
```

表 1_2 给出 OpenGL ES 命令后缀和输入参数类型的描述：

缩写	数据类型	对应的 C 语言类型	GL 类型
b	8-bit signed interger	signed char	Glbyte
ub	8-bit unsigned interger	unsigned char	Glubyte
			Glboolean
s	16-bit signed interger	short	Glshort
us	16-bitu signed interger	unsigned short	Glushort
I	32-bit signed interger	int	Glint
Ui	32-bit unsigned interger	unsigned char	Gluint
			Glbitfield
			Glenum
X	16-bit fixed point	int	Glfiexed
F	32-bit fixed point	float	Glfloat
			Glclampf

最后，OpenGL ES 定义 Glvoid 数据类型，作为 OpenGL ES 接受的指针类型。

本书最后，给出了 OpenGL ES 命令的参考列表，*被使用代表命令的多个版本，例如 glUniform*()可使用所有指定的变量，glUniform*v()你能使用命令的矢量版本，如果谈到特殊的命令版本，我们将使用命令全名和对应的后缀。

出错处理

OpenGL ES 命令出错会产生一个错误码，错误码被记录，能够使用 glGetError 函数查询，在第一个被记录的错误码被查询前，不会记录新的错误码。一旦错误码被查询，当前错误码将变成 GL_NO_ERROR。除了 GL_OUT_OF_MEMORY 错误码以外，其它的错误码将被忽略，不影响程序运行状态。

```
GLenum     glGetError(void)
```

返回当前错误码，并将错误码复位成 GL_NO_ERROR，如果返回 GL_NO_ERROR，说明自从上次查询后，没有错误产生。

表 1_3 列出基本的错误码及描述，除此以外的错误码被描述在各自的命令章节。

表 1-3 OpenGL ES 基本错误码

错误码	描述
GL_NO_ERROR	没错误产生自从上次 glGetError()调用后
GL_INVALID_ENUM	枚举值超出枚举范围，错误被忽略
GL_INVALID_VALUE	数值超出枚举范围，错误被忽略
GL_INVALID_OPERATION	命令不能被执行，在当前环境下，错误被忽略
GL_OUT_OF_MEMORY	没有足够内存执行命令，如果错误产生，OpenGL ES 的管线状态不确定

Flush 和 Finish

OpenGL ES 2.0 API 继承自 OpenGL 的客户机-服务器模式。应用程序或者叫客户机发出指令，命令在执行机或者叫服务器上执行。OpenGL 中，客户机和服务器可以是一个网络中的不同机器，但 OpenGL ES 适用于嵌入式及便携平台，一般情况下客户机和服务器是同一

个设备。

客户机-服务器模式下，被客户机发出的命令不需要服务器的立即回复，如果客户机和服务器在一个网络中，一次送一个命令将显得效率低下，因此命令将在客户机缓存，随后一起发送，因此客户机需要一个机制确认服务器已经完成了命令的执行，考虑到 OpenGL ES 是多线程的，为同步各个线程的执行，例如线程 B 的执行依靠线程 A 的执行结果，线程 A 的执行情况将是重要的。Flush 命令对服务器发出执行指令，但不等待指令执行完成，如果需要等待指令完成，需要使用 Finish 指令。一般不赞成使用 Finish，除非必须，Finish 会等待客户机将命令执行完成再返回，它强迫客户机和服务器之间同步执行。

void glFlush(void)

void glFinish(void)

基本状态管理

图 1-1 显示 OpenGL ES 2.0 各种管线状态，每种状态能够被使能或关闭，状态值被各自的线程维护，例如抖动、剔除、混合等等。当 OpenGL ES 上下文初始化时，这些状态被初始化。状态使能可以使用 glEnable 和 glDisable 命令设置。

void glEnable(GLenum cap)

void glDisable(GLenum cap)

glEnable 和 glDisable 使能或关闭各种功能，初始化时除了 GL_DITHER 功能设置为 GL_TRUE 外，其它功能都设置为 GL_FALSE，如果输入非法参数，产生 GL_INVALID_ENUM 错误码。

下面的章节将描述图 1-1 中管线的各种状态，你能够使用 glIsEnabled 函数查询某个状态当前是否使能。

GLboolean glIsEnabled(GLenum cap)

返回 GL_TRUE 或 GL_FALSE 依据当前状态是否被使能或不使能，如果输入非法参数，产生 GL_INVALID_ENUM 错误码。

特殊的状态值像混合因子，深度测试值等也使用 glGet***函数查询，这些命令在 14 章状态描述一节详细描述。

扩展阅读

OpenGL ES 1.0, 1.1, 和 2.0 的技术规格书能够在 www.khronos.org/opengles/找到，Khronos 的网站(www.khronos.org)有最新的 Khronos 技术规格书、开发者论坛、教程和例子。

1. Khronos OpenGL ES 1.1 Web site: www.khronos.org/opengles/1_X/
2. Khronos OpenGL ES 2.0 Web site: www.khronos.org/opengles/2_X/
3. Khronos EGL Web site: www.khronos.org/egl/

2 Hello Triangle：一个 OpenGL ES 2.0 例子

开始介绍 OpenGL ES 2.0 基本概念前，我们先看一个简单的例子。本章我们先看一个比较简单的绘制三角形的 OpenGL ES 2.0 例子。我们将写的和绘制其他的几何图形是非常类似的。这一章主要讲到这些内容：

使用 EGL 创造一个显示渲染窗口平面。

装载顶点和片段着色器。

创造一个项目，联系顶点和片段着色器，链接项目。

设置视窗。

清除颜色缓冲区。

最基本的渲染。

在 EGL 窗口显示颜色缓冲区的内容

在开始绘制 OpenGL ES 2.0 三角形前，有几个重要的步骤。本章简单阐述这些基本步骤，后面的章节详细介绍这些步骤，深入讲述 API 内容，我们的目的是使你有初步印象怎样运行一个最简单的 OpenGL ES 2.0 应用。

代码架构

这本书，我们建立一个使用 OpenGL ES 2.0 编程的运行函数库。通过这本书的例子，我们的目标是：

1. 这应该是简单的、短的、容易理解的。我们关心的例子代码不是代码本身的长度量，而是 OpenGL ES 2.0 的函数使用，因此我们的代码是简单易懂，目的是了解 OpenGL ES 2.0 API 观念。
2. 容易移植。虽然我们在微软系统开始代码，但我们也想我们的例子容易移植到其他平台，我们使用 C 语言而不是 C++，因为 C++在不同的平台有不同限制，我们避免使用全局数据，因为在一些平台不允许使用。

本书中我们介绍了一些代码，附录 D 中你能够找到全部的代码，

如何下载例子

你能在下面的网站下载本书的例子

www.opengles-book.com.

例子运行在微软的 XP 和 Vista 系统，其中显卡（GPU）需要支持 OpenGL 2.0，例子编程环境是 VS2005.能够在 AMD 的 OpenGL ES 2.0 模拟器中编译运行，着色器的例子使用 AMD 的 RenderMonkey 着色器工具执行。上面的网站提供这个工具的下载，以上工具都是免费的，读者没有 VS 工具的话也可以使用 VS2008 的免费版，在 www.microsoft.com/express/下载。

Hello Triangle 例子

让我们看一下 Hello Triangle 例子，例 2-1，对比熟悉固定功能管线例子的读者，可能想怎么有这么多代码，对不熟悉桌面 OpenGL 的读者来说，可能认为画一个三角形怎么有这么多代码。因为 OpenGL ES 2.0 是以可编程着色器为基础的，这意味着你绘制任何图形都必须有一个合适的着色器装载和绑定，比使用固定管线的桌面版本有更多代码。

Example 2-1 Hello Triangle 例子

```
#include "esUtil.h"
typedef struct
{
    // Handle to a program object
    GLuint programObject;
} UserData;
///
```

```
// Create a shader object, load the shader source, and
// compile the shader.
//
GLuint LoadShader(const char *shaderSrc, GLenum type)
{
    GLuint shader;
    GLint compiled;

    // Create the shader object
    shader = glCreateShader(type);
    if(shader == 0)
        return 0;
    // Load the shader source
    glShaderSource(shader, 1, &shaderSrc, NULL);

    // Compile the shader
    glCompileShader(shader);
    // Check the compile status
    glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
    if(!compiled)
    {
        GLint infoLen = 0;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);

        if(infoLen > 1)
        {
            char* infoLog = malloc(sizeof(char) * infoLen);
            glGetShaderInfoLog(shader, infoLen, NULL, infoLog);
            esLogMessage("Error compiling shader:\n%s\n", infoLog);
            free(infoLog);
        }
        glDeleteShader(shader);
        return 0;
    }
    return shader;
}
///
// Initialize the shader and program object
//
int Init(ESContext *esContext)
{
    UserData *userData = esContext->userData;
    GLbyte vShaderStr[] =
        "attribute vec4 vPosition;      \n"
```

```
    "void main()                        \n"
    "{                                  \n"
    "     gl_Position = vPosition; \n"
    "}                                  \n";

GLbyte fShaderStr[] =
    "precision mediump float;                       \n"
    "void main()                                    \n"
    "{                                              \n"
    "   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); \n"
    "}                                              \n";
GLuint vertexShader;
GLuint fragmentShader;
GLuint programObject;
GLint linked;
// Load the vertex/fragment shaders
vertexShader = LoadShader(GL_VERTEX_SHADER, vShaderStr);
fragmentShader = LoadShader(GL_FRAGMENT_SHADER, fShaderStr);
// Create the program object
programObject = glCreateProgram();
if(programObject == 0)
    return 0;
glAttachShader(programObject, vertexShader);
glAttachShader(programObject, fragmentShader);
// Bind vPosition to attribute 0
glBindAttribLocation(programObject, 0, "vPosition");
// Link the program
glLinkProgram(programObject);
// Check the link status
glGetProgramiv(programObject, GL_LINK_STATUS, &linked);
if(!linked)
{
    GLint infoLen = 0;
    glGetProgramiv(programObject, GL_INFO_LOG_LENGTH, &infoLen);

    if(infoLen > 1)
    {
        char* infoLog = malloc(sizeof(char) * infoLen);
        glGetProgramInfoLog(programObject, infoLen, NULL, infoLog);
        esLogMessage("Error linking program:\n%s\n", infoLog);

        free(infoLog);
    }
    glDeleteProgram(programObject);
```

```
        return FALSE;
    }
    // Store the program object
    userData->programObject = programObject;
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    return TRUE;
}
///
// Draw a triangle using the shader pair created in Init()
//
void Draw(ESContext *esContext)
{
    UserData *userData = esContext->userData;
    GLfloat vVertices[ ] = {0.0f,  0.5f, 0.0f,
                            -0.5f, -0.5f, 0.0f,
                            0.5f, -0.5f,  0.0f};

    // Set the viewport
    glViewport(0, 0, esContext->width, esContext->height);

    // Clear the color buffer
    glClear(GL_COLOR_BUFFER_BIT);
    // Use the program object
    glUseProgram(userData->programObject);
    // Load the vertex data
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vVertices);
    glEnableVertexAttribArray(0);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    eglSwapBuffers(esContext->eglDisplay, esContext->eglSurface);
}
int main(int argc, char *argv[ ])
{
    ESContext esContext;
    UserData  userData;
    esInitialize(&esContext);
    esContext.userData = &userData;
    esCreateWindow(&esContext, "Hello Triangle", 320, 240,
                   ES_WINDOW_RGB);

    if(!Init(&esContext))
        return 0;
    esRegisterDrawFunc(&esContext, Draw);

    esMainLoop(&esContext);
```

}

编译运行例子

本书的例子都运行在 AMD 的 OpenGL ES 2.0 模拟器上，这个模拟器提供能够执行 EGL 1.3 和 OpenGL ES 2.0 APIs 的窗口。Khronos 提供的 GL2 和 EGL 标准头文件被使用连接模拟器的接口。模拟器可以完全仿真 OpenGL ES 2.0，因此在模拟器运行的代码可以无缝的运行在实际的设备中，模拟器要求你的显卡支持桌面版 OpenGL 2.0。

我们的代码可以方便的移植到其他平台，然而本书的例子都是使用 32 位系统上的 VS2005 编译，运行于 AMD's OpenGL ES 2.0 模拟器。OpenGL ES 2.0 例子路径这样组织：

Common/ 包含着 OpenGL ES 2.0 程序架构设置、代码、和模拟器。

Chapter_X/ 包含每章的例子程序，一个 VS2005 的项目解决方案被提供。

为编译运行例子，使用 VS2005 打开 Chapter_2/Hello_Triangle/Hello_Triangle.sln，例子将被编译和运行，你将看到图 2-1 所示的图像。

Figure 2-1 Hello Triangle Example

主要本书的后面章节提供了一些使用 RenderMonkey v1.80 的着色器程序例子，RenderMonkey 提供灵活的开发着色器程序的集成开发环境，哪些带.rfx 后缀的文件能够 RenderMonkey 使用。RenderMonkey 的屏幕截图如颜色调色板 2 所示。

使用 OpenGL ES 2.0 架构

在 main 函数里，你能看到几个 ES 函数，主函数的第一件事情是定义一个 ESContext，并初始化它，

```
ESContext esContext;
UserData  userData;
esInitialize(&esContext);
esContext.userData = &userData;
```

本书的每个例子都要做同样的事情，ESContext 被所有的 ES 架构引入，它包含着 ES 架构需要的所有必要信息。使用 ESContext 的原因为 ES 代码可以不再必须使用全局数据。

很多平台不允许程序声明全局数据，例如 BREW 和 Symbian 因此我们使用 ESContext 避免使用全局变量。

ESContext 有一个名为 userData 的成员变量，它的类型为 void *指针，每个程序都用它来存储数据，esInitialize 用来初始化 context 和 ES 代码架构。其它的 ESContext 结构成员在

头文件中描述，程序只能读取这些值，这些结构成员包括窗口宽和高、EGLcontext、回调函数指针等。

Main 函数接下来的工作是创建窗口，初始化绘制回调函数、进入主循环。

```
esCreateWindow(&esContext, "Hello Triangle", 320, 240,ES_WINDOW_RGB);
if(!Init(&esContext))
    return 0;
esRegisterDrawFunc(&esContext, Draw);

esMainLoop(&esContext);
```

esCreateWindow 调用创造一个指定宽度和高度的窗口（例子中是 320 × 240），最后一个参数是指定窗口创造的位数，是可选相，在例子中我们使用的是 RGB 帧缓冲。第三章，EGL 介绍，我们讨论 esCreateWindow 函数的更多细节，这个函数使用 EGL 创造一个可显示联系着窗口的平面，EGL 是独立于平台的创造渲染平面和上下文环境的 API，细节以后详述。

调用 esCreateWindow 函数后，下一件事情是初始化哪些对项目必须的东西，然后再注册一个 Draw 的回调函数，这将引起窗口渲染开始，最后程序进入主循环，直到项目窗口被关闭。

创建一个简单的顶点和片段着色器

OpenGL ES 2.0 中，在有效的顶点和片段着色器被装载前，什么渲染都做不了。章节一中我们介绍管线时介绍了顶点和片段着色器，做任何渲染前，必须有顶点和片段着色器。

初始化最大的任务是装载顶点和片段着色器，顶点着色器在项目中是很简单的：

```
GLbyte vShaderStr[ ] =
    "attribute vec4 vPosition;     \n"
    "void main()                   \n"
    "{                             \n"
    "     gl_Position = vPosition; \n"
    "};                            \n";
```

顶点着色器定义一个输入 attribute，它是 4 个成员的矢量 vPosition。后面的 Draw 函数将为每个顶点设置位置变量值。主函数声明着色器宣布着色器开始执行。着色器主体非常简单，它复制输入 vPosition 属性到 gl_Position 输出变量中，每个顶点着色器必须输出位置值到 gl_Position 变量中，这个变量传入到管线的下一个阶段中，着色器编程是本书最主要的内容，但本节仅仅给出顶点着色器基本样子。第 5 章 OpenGL ES 着色器语言，讲述着色器语法，第 8 章 顶点着色器，讲述怎么写顶点着色器。

片段着色器也非常简单：

```
GLbyte fShaderStr[] =
    "precision mediump float;                  \n"
    "void main()                               \n"
    "{                                         \n"
    "   gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); \n"
    "}                                         \n";
```

第一行宣布着色器默认的浮点变量精度，更详细的说明看第 5 章精度限定部分，现在我们注意力放在 main 函数上，它的输出值(1.0, 0.0, 0.0, 1.0)赋给变量 gl_FragColor，gl_FragColor 是片段着色器最终的输出值，本例中输出值是红色，片段着色器的编程细节在第 9 章叙述，第 10 章介绍贴图，这里仅介绍一个大概。

典型的一个游戏或应用将不会内联一个着色器源码串，大多数的实际应用着色器应该填

充文字或数据，然后被 API 装载，我们为简化起见，在程序源码上直接赋值。

编译和装载着色器

定义了着色器源码后，我们将着色器装入 OpenGL ES，这由 LoadShader 函数完成，检查没有错误后，这个函数返回着色器对象，这个对象在后面被用于连接项目对象，这两个对象在第四章着色器和项目中描述。

让我们看一下 LoadShader 函数，首先使用 glCreateShader 创建着色器对象，它语法如下：

```
GLuint LoadShader(GLenum type, const char *shaderSrc)
{
    GLuint shader;
    GLint compiled;

    // Create the shader object
    shader = glCreateShader(type);
    if(shader == 0)
    return 0;
```

使用 glShaderSource 装载着色器源码，使用 glCompileShader 装载着色器对象。

```
// Load the shader source
 glShaderSource(shader, 1, &shaderSrc, NULL);

 // Compile the shader
 glCompileShader(shader);
```

装载着色器后，装载状态和产生的错误被打印出来：

```
  // Check the compile status
    glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
    if(!compiled)
    {
        GLint infoLen = 0;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);

        if(infoLen > 1)
        {
            char* infoLog = malloc(sizeof(char) * infoLen);
            glGetShaderInfoLog(shader, infoLen, NULL, infoLog);
            esLogMessage("Error compiling shader:\n%s\n", infoLog);

            free(infoLog);
        }
        glDeleteShader(shader);
        return 0;
    }
    return shader;
}
```

如果着色器对象装载成功，一个新的着色器对象被返回，在后面将连接到项目对象上，细节在第 4 章开始部分讲述。

创建项目对象链接着色器

一旦应用程序已经创建了顶点、片段着色器对象，它需要去创建项目对象，项目是最终的链接对象，每个着色器在被绘制前都应该联系到项目或者项目对象。

创建项目对象和链接在第 4 章介绍，这里简单说明，第一步创建项目对象，链接顶点着色器和片段着色器。

```
// Create the program object
programObject = glCreateProgram();
if(programObject == 0)
    return 0;
glAttachShader(programObject, vertexShader);
glAttachShader(programObject, fragmentShader);
```

两个着色器被链接后，下一步应用设定顶点着色器 vPosition 属性：

```
// Bind vPosition to attribute 0
glBindAttribLocation(programObject, 0, "vPosition");
```

第 6 章，顶点属性、顶点矩阵、缓冲区对象，讲述更多的细节现在我们看 glBindAttribLocation 函数绑定 vPosition 属性到顶点着色器位置 0，当我们指定顶点数据后，位置指针指向下一个位置。

最后我们链接项目检查错误。

```
// Link the program
glLinkProgram(programObject);
// Check the link status
glGetProgramiv(programObject, GL_LINK_STATUS, &linked);
if(!linked)
{
    GLint infoLen = 0;
    glGetProgramiv(programObject, GL_INFO_LOG_LENGTH, &infoLen);

    if(infoLen > 1)
    {
        char* infoLog = malloc(sizeof(char) * infoLen);
        glGetProgramInfoLog(programObject, infoLen, NULL, infoLog);
        esLogMessage("Error linking program:\n%s\n", infoLog);

        free(infoLog);
    }
    glDeleteProgram(programObject);
    return FALSE;
}
// Store the program object
userData->programObject = programObject;
```

所有这些步骤后，编译着色器，检查错误，创建项目对象，附加上着色器，链接项目，检查链接错误。成功后可以使用项目对象去渲染。为使用项目对象去渲染，我们使用 glUseProgram 绑定。

```
// Use the program object
```

glUseProgram(userData->programObject);

使用 glUseProgram 调用项目句柄后，所有的以后的渲染将使用顶点着色器和片段着色器去联系项目对象。

设定窗口和清除缓冲区

我们已经用 EGL 创建了渲染平面初始化并装载了着色器，准备去绘制实际的物体，Draw 回调函数绘制帧。我们解释的 Draw 第 1 个函数是 glViewport，它定义 OpenGL ES 的将要绘制物体的 2D 窗口的坐标原点和窗口宽度和高度。在 OpenGL ES 中 viewport 定义渲染操作将要显示的 2D 长方形显示区域。

// Set the viewport

glViewport(0, 0, esContext->width, esContext->height);

viewport 设定窗口的原点 origin (x, y)、宽度和高度，在第 7 章基本装配和光栅化这一节的坐标系统和剪切中介绍 glViewport 细节。

设定窗口后，下一步是清除屏幕，在 OpenGL ES 中，有多种需要绘制的缓冲区类型，颜色、深度和模板。在第 11 章片段操作，我们讨论哪些缓冲区的细节。在本例中仅仅颜色缓冲区被使用。开始每帧绘制前，我们使用 glClear 清除有时缓冲区。

// Clear the color buffer

glClear(GL_COLOR_BUFFER_BIT);

缓冲区将被用 glClearColor 函数的颜色参数值清除，本例初始化结束后，清除颜色被设定为(0.0, 0.0, 0.0, 1.0)，于是屏幕变成黑色。清除颜色通过 glClear 设定。

装载几何图像绘制基元

现在我们已清除了颜色缓冲区，设定了视口，装载了项目对象，我们指定要绘制的几何图形是三角形，三角形顶点的坐标通过 vVertices 矩阵设定设定三个(x, y, z)坐标。

```
GLfloat vVertices[] = {0.0f,  0.5f, 0.0f,
                       -0.5f, -0.5f, 0.0f,
                       0.5f, -0.5f, 0.0f};
…
// Load the vertex data
glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vVertices);
glEnableVertexAttribArray(0);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

顶点位置需要被装载到 GL 联系到 vPosition，你是否想到先前我们绑定 vPosition 变量到属性位置 0，每个顶点着色器中的属性都有一个唯一的用无符号整形数标示的位置，调用 glVertexAttribPointer 函数，我们把数据装载到位置 0，第六章我们详述怎样装载顶点属性和使用顶点矩阵。

绘制三角形最后一步是调用 OpenGL ES 去绘制基元，本例中使用 glDrawArrays 函数。这个函数绘制三角形、直线或带状物等基元，将在第 7 章详细介绍这些几何图形的细节。

在后缓冲区显示

最后在谈谈绘制在缓冲区中的三角形。最后说明缓冲区如何显示到屏幕上，讨论之前，先介绍一点双缓冲区概念。

在显示屏上看到的帧缓冲区是 2 维的空间的像素数据，可能想到的方法是简单更新可见缓冲区中的数据，直接更新显示缓冲区中数据有困难，对显示屏幕缓冲区内存更新要有固定的频率。如果两次时间不同，将看到闪烁的痕迹。

为解决这个问题，系统一般使用双缓冲区：前缓冲区和后缓冲区，所有的渲染发生在后缓冲区，它是屏幕上不可见的缓冲区内存，渲染完成后，交换前后缓冲区，即在下一帧时前

缓冲区变成后缓冲区。

使用这个技术，在一帧渲染完成前，我们不显示任何图像，这种 OpenGL ES 方法使用通过 EGL 实现，使用函数是 eglSwapBuffers;

```
eglSwapBuffers(esContext->eglDisplay, esContext->eglSurface);
```

这个 EGL 函数交换前后缓冲区，eglSwapBuffers 的输入参数是 EGL 显示区和窗口，这两个参数代表了实际的物理显示区和渲染区。下一章我们解释这个函数的更多细节还有清除渲染区物体、上下文、缓冲区管理。现在我们知道我们交换缓冲区后，显示了要显示的三角形。

3 EGL 介绍

第 2 章我们使用 OpenGL ES 2.0 在窗口绘制了一个三角形，但我们使用自己的函数去打开和管理窗口，虽然我们的例子很简单，但是它让 OpenGL ES 2.0 在你的系统工作时减少你的工作量。

Khronos 为发展编程上下文环境，提供了平台独立的 API 叫作 EGL，用来管理绘制窗口，（微软视窗只是一种，后面将谈到其它的）EGL 提供下面的机制。

你使用的系统窗口之间的通讯

查询可用的类型，配置绘图窗口

创建绘图窗口

同步 OpenGL ES 2.0 渲染和其它绘图 API 的渲染（例如 OpenVG，或者你系统的其它绘图命令）

管理渲染资源像贴图纹理

本章介绍基本创建窗口的要求，还要其它操作，像创建贴图纹理、使用 EGL 命令的要求等。

窗口系统之间通讯

EGL 提供 OpenGL ES 2.0（包括其它 Khronos 工作组的 API）和你计算机运行的操作系统之间通讯，例如运行 X 视窗的 GNU/Linux 系统、微软系统、苹果的 Mac OS 系统。EGL 在决定绘制什么类型的窗口前，需要打开和操作系统的通讯连接。

每个操作系统有不同的语法，EGL 提供一个基本的不透明的类—EGLDisplay—它封装了与操作系统相关的连接。使用 EGL 第一步是创建并初始化一个使用本操作系统 EGL 显示的连接。这需要两步，看例 3-1：

例 3-1 初始化 EGL

```
EGLint    majorVersion;
EGLint    minorVersion;
EGLDisplay    display;
display = eglGetDisplay(EGL_DEFAULT_DISPLAY);
if(display == EGL_NO_DISPLAY)
{
    // Unable to open connection to local windowing system
}
if(!eglInitialize(display, &majorVersion, &minorVersion))
{
    // Unable to initialize EGL. Handle and recover
}
```

打开一个 EGL 显示服务连接使用：

```
EGLDisplay eglGetDisplay(EGLNativeDisplayType display_id);
```

EGLNativeDisplayType 定义使用者的操作系统，对微软是个 HDC—微软操作系统设备上下文句柄。对应用程序来说这是很容易的移植到其他的操作系统，使用 EGL_DEFAULT_DISPLAY 将建立一个操作系统联系。

如果连接没有建立，eglGetDisplay 将返回 EGL_NO_DISPLAY，这个错误指示 EGL 不可用，你将不能使用 OpenGL ES 2.0。

开始操作更多的 EGL 行为前，你需要一个简明的 EGL 的描述处理你应用程序的错误。

检查错误

大多数情况下 EGL 返回 EGL_TRUE 或者 EGL_FALSE，当然可以得到更多的信息，如果返回 EGL_FALSE，我们能够查询发生错误的原因，你能够查询错误码，使用

```
EGLint eglGetError();
```

你可能疑惑为什么不直接返回错误码，我们不支持忽略返回码，虽然这样做减少了程序代码设计的冗余，但你能够在代码开发和调试阶段查询查询错误码，当你完成了你的程序设计，你可以减少你的错误检查。

初始化 EGL

当你成功的创建了一个连接，接下来需要初始化：

```
EGLBoolean eglInitialize(EGLDisplay display, EGLint *majorVersion, EGLint *minorVersion);
```

初始化 EGL 内部数据，返回 EGL 的主次版本号。如果 EGL 不能初始化，它将返回 EGL_FALSE，设定错误码如下：

如果显示指定的不是合法的 EGLDispla，返回 EGL_BAD_DISPLAY

如果 EGL 不能初始化，返回 EGL_NOT_INITIALIZED

设定可用的窗口配置

初始化 EGL 后，我们需要设定渲染何种类型的窗口和配置信息。这需要两步。

1. 查询窗口配置，找到最好的选择
2. 指定要求，让 EGL 做出做最好的匹配。

大多数情况下第 2 步和第 1 步是类似的。或者 EGL 将返回一个 EGLConfig 结构，它标示了窗口和它特定的信息，像颜色位数、深度缓冲区，你能使用 eglGetConfigAttribute 函数查询 EGLConfig 的属性。

查询微软的窗口配置信息使用：

```
EGLBoolean eglGetConfigs(EGLDisplay display, EGLConfig *configs,
                         EGLint maxReturnConfigs,EGLint *numConfigs);
```

如果成功返回 EGL_TRUE。

有两种方式调用 eglGetConfigs，输入参数如果是 NULL，系统将返回 EGL_TRUE，并设定可用的 EGLConfigs 数目 numConfigs，但不包含其它信息，但找到了可用的 EGLConfigs 数目，你就可以分配合适的内存数目来存储整个 EGLConfigs。

或者你输入一个 EGLConfig 结构体，以它为参数调用 eglGetConfigs，设定 maxReturnConfigs 为你分配的矩阵大小，即将返回 configs 的最大数。调用完成后，numConfigs 是更新后的 configs 数，你可以处理返回值列表，查询配置特定决定怎么匹配我们的需要。

查询 EGLConfig 属性

我们使用 EGLConfig 标示 EGL 的结构，你能查询它的值。

一个 EGLConfig 标示了被 EGL 使用的各种窗口信息，包括颜色缓冲区、深度缓冲区、模板缓冲区、窗口类型及其它。EGLConfig 是一个可供查询的属性列表，本章我们仅讨论它的一个子集，整个列表在附录 3-1 列出。

查询某个属性使用：

```
EGLBoolean eglGetConfigAttrib(EGLDisplay display, EGLConfig config,
                              EGLint attribute, EGLint *value)
```

它将返回你查询的属性，这将让你控制你选择的属性创建渲染窗口，这可能对可选项迷惑，

你可以使用 eglChooseConfig 去找出最匹配你要求的窗口。

表 3-1 EGLConfig 属性

Attribute	Description	Default Value
EGL_BUFFER_SIZE	颜色缓冲区中所有组成颜色的位数	0
EGL_RED_SIZE	颜色缓冲区中红色位数	0
EGL_GREEN_SIZE	颜色缓冲区中绿色位数	0
EGL_BLUE_SIZE	颜色缓冲区中蓝色位数	0
EGL_LUMINANCE_SIZE	颜色缓冲区中亮度位数	0
EGL_ALPHA_SIZE	颜色缓冲区中透明度位数	0
EGL_ALPHA_MASK_SIZE	遮挡缓冲区透明度掩码位数	0
EGL_BIND_TO_TEXTURE_RGB	绑定到 RGB 贴图使能为真	EGL_DONT_CARE
EGL_BIND_TO_TEXTURE_RGBA	绑定到 RGBA 贴图使能为真	EGL_DONT_CARE
EGL_COLOR_BUFFER_TYPE	颜色缓冲区类型 EGL_RGB_BUFFER, 或者 EGL_LUMINANCE_BUFFER	EGL_RGB_BUFFER
EGL_CONFIG_CAVEAT	配置有关的警告信息	EGL_DONT_CARE
EGL_CONFIG_ID	唯一的 EGLConfig 标示值	EGL_DONT_CARE
EGL_CONFORMANT	使用 EGLConfig 创建的上下文符合要求时为真	—
EGL_DEPTH_SIZE	深度缓冲区位数	0
EGL_LEVEL	帧缓冲区水平	0
EGL_MAX_PBUFFER_WIDTH	使用 EGLConfig 创建的 PBuffer 的最大宽度	—
EGL_MAX_PBUFFER_HEIGHT	使用 EGLConfig 创建的 PBuffer 最大高度	—
EGL_MAX_PBUFFER_PIXELS	使用 EGLConfig 创建的 PBuffer 最大尺寸	—
EGL_MAX_SWAP_INTERVAL	最大缓冲区交换间隔	EGL_DONT_CARE
EGL_MIN_SWAP_INTERVAL	最小缓冲区交换间隔	EGL_DONT_CARE
EGL_NATIVE_RENDERABLE	如果操作系统渲染库能够使用 EGLConfig 创建渲染渲染窗口	EGL_DONT_CARE
EGL_NATIVE_VISUAL_ID	与操作系统通讯的可视 ID 句柄	EGL_DONT_CARE
EGL_NATIVE_VISUAL_TYPE	与操作系统通讯的可视 ID 类型	EGL_DONT_CARE
EGL_RENDERABLE_TYPE	渲染窗口支持的布局组成标示符的遮挡位 EGL_OPENGL_ES_BIT, EGL_OPENGL_ES2_BIT, or EGL_OPENVG_BIT that	EGL_OPENGL_ES_BIT
EGL_SAMPLE_BUFFERS	可用的多重采样缓冲区位数	0
EGL_SAMPLES	每像素多重采样数	0

EGL_STENCIL_SIZE	模板缓冲区位数	0
EGL_SURFACE_TYPE	EGL 窗口支持的类型 EGL_WINDOW_BIT, EGL_PIXMAP_BIT, 或 EGL_PBUFFER_BIT	EGL_WINDOW_BIT
EGL_TRANSPARENT_TYPE	支持的透明度类型	EGL_NONE
EGL_TRANSPARENT_RED_VALUE	透明度的红色解释	EGL_DONT_CARE
EGL_TRANSPARENT_GREEN_VALUE	透明度的绿色解释	EGL_DONT_CARE
EGL_TRANSPARENT_BLUE_VALUE	透明度的兰色解释	EGL_DONT_CARE

让 EGL 选择配置

使用

```
EGLBoolean eglChooseChofig(EGLDispay display, const EGLint *attribList,
                           EGLConfig *config, EGLint maxReturnConfigs,
                           ELGint *numConfigs );
```

你要提供属性列表，所有相关的最优值对你的应用程序正确运行可能非常重要。例如你想让你的程序创建的窗口支持 5 位红蓝、6 位绿色（RGB565 格式），你能像例 3-2 那样设置。

例-2 配置 EGL 属性

```
EGLint attribList[] =
{
    EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
    EGL_RED_SIZE, 5,
    EGL_GREEN_SIZE, 6,
    EGL_BLUE_SIZE, 5,
    EGL_DEPTH_SIZE, 1,
    EGL_NONE
};
```

对属性列表中没有明确指定的值，EGL 使用表 3-1 的默认值，指定一个属性值，EGL 将保证返回一个可以的最小的匹配。

例 3-3 演示怎么使用选择的属性值

例 3-3 查询窗口配置

```
const EGLint MaxConfigs = 10;
EGLConfig    configs[MaxConfigs];   // We'll only accept 10 configs
EGLint    numConfigs;
if(!eglChooseConfig(dpy, attribList, configs, MaxConfigs,
    &numConfigs))
{
    // Something didn't work … handle error situation
}
else
{
    // Everything's okay. Continue to create a rendering surface
}
```

如果 eglChooseConfig 返回成功，一系列的符合你要求的 EGLConfigs 将返回，超过一个的

返回值时 eglChooseConfig 按下面的顺序选择配置

1. 如果没有警告值（EGL_CONFIG_CAVEAT 是 GL_NONE），EGL_CONFIG_CAVEAT 优先配置，然后是慢速渲染配置（EGL_SLOW_CONFIG)，最后是非一致配置（EGL_NON_CONFORMANT_CONFIG）。
2. 缓冲区类型通过 EGL_COLOR_BUFFER_TYPE 指定。
3. 颜色缓冲区地址下降模式，缓冲区位数依靠 EGL_COLOR_BUFFER_TYPE 指定的类型，将是最小的指定的颜色通道值。缓冲区类型是 EGL_RGB_BUFFER，位数依据 EGL_RED_SIZE、 EGL_GREEN_SIZE 和 EGL_BLUE_SIZE 的总和。当颜色缓冲区类型是 EGL_LUMINANCE_BUFFER，位数是 EGL_LUMINANCE_SIZE 和 EGL_ALPHA_SIZE。
4. EGL_BUFFER_SIZE 升序排列
5. EGL_SAMPLE_BUFFERS 值为升序
6. EGL_SAMPLES 数值升序
7. EGL_DEPTH_SIZE 值升序
8. EGL_STENCIL_SIZE 值升序
9. EGL_ALHPA_MASK_SIZE 值（仅仅用于 OpenVG 窗口）
10. EGL_NATIVE_VISUAL_TYPE 是一个独立于使用者的坚持开发环境
11. EGL_CONFIG_ID 升序

没有在列表中提到的参数不能使用排序。

像例子中提到的，如果 eglChooseConfig 返回成功，我们就有足够的信息去创建窗口并绘制图形。否则你不能指定你要渲染的窗口类型，（指定使用 EGL_SURFACE_TYPE）EGL 认为你要创建显示可见的窗口。

创建在屏显示渲染区：EGL 窗口

一旦我们有一个合适的 EGLConfig，可以调用下面函数创建窗口：

```
EGLSurface eglCreateWindowSurface(EGLDisplay display,
                                  EGLConfig config,
                                  EGLNatvieWindowType window,
                                  const EGLint *attribList)
```

这个函数让程序联系到操作系统的窗口管理，窗口参数为 EGLConfig，另外它要求操作系统先前已经创建一个窗口。EGL 是操作系统和 OpenGL ES 2.0 之间的软件层，说明如何创建操作系统窗口超出了本书范围，请参考你的操作系统书籍决定如何创建满足你系统的窗口。

最后，还需要一个属性列表，这个列表不同于表 3-1，因为 EGL 也支持其它的渲染 API（如 OpenVG）那些属性 eglCreateWindowSurface 可以接受，但 OpenGL ES 2.0（表 3-2）却不能使用，对我们来说，eglCreateWindowSurface 接收一个单一的属性，就是前或者后缓冲区。

表 3-2 使用 eglCreateWindowSurface 创建的窗口属性

Token	Description	Default Value
EGL_RENDER_BUFFER	指定哪个缓冲区被用于渲染(EGL_FRONT_BUFFER 或 EGL_BACK_BUFFER).	EGL_BACK_BUFFER

主意：OpenGL ES 2.0 仅支持双缓冲区

属性列表可以是空（允许空指针作为 attribList 值），或者可以是首个元素为 EGL_NONE 的列表，那么使用相应属性的默认值。

由于某些原因调用 eglCreateWindowSurface 会失败，这时函数调用返回

EGL_NO_SURFACE，我们可以使用 eglGetError 查询错误发生原因，可能的原因如表 3-3：

表 3-3　eglCreateWindowSurface 失败可能的原因

Error	CodeDescription
EGL_BAD_MATCH	系统窗口属性不匹配 EGLConfig 值 EGLConfig 值不支持渲染窗口（例如 EGL_SURFACE_TYPE 属性不是 EGL_WINDOW_BIT 设置的值）
EGL_BAD_CONFIG	EGLConfig 不被系统支持
EGL_BAD_NATIVE_WINDOW	系统窗口句柄无效
EGL_BAD_ALLOC	eglCreateWindowSurface 不能分配新窗口的资源或者已存在 EGLConfig 的系统窗口

综上所述，我们创建窗口的代码如例 3-4 所示：

例 3-4 创建 EGL 窗口

```
EGLRenderSurface    window;
EGLint attribList[] =
{
    EGL_RENDER_BUFFER, EGL_BACK_BUFFER,
    EGL_NONE
);
window = eglCreateWindowSurface(dpy, window, config, attribList);
if(window == EGL_NO_SURFACE)
{
    switch(eglGetError())
    {
        case EGL_BAD_MATCH:
            // Check window and EGLConfig attributes to determine
            // compatibility, or verify that the EGLConfig
            // supports rendering to a window,
            break;
          case EGL_BAD_CONFIG:
            // Verify that provided EGLConfig is valid
            break;
        case EGL_BAD_NATIVE_WINDOW:
            // Verify that provided EGLNativeWindow is valid
            break;
        case EGL_BAD_ALLOC:
            // Not enough resources available. Handle and recover
            break;
    }
}
```

绘制了要做图的窗口，在使用 OpenGL ES 2.0 在窗口绘制前，还有两步要做。可见窗口不是我们唯一我们能够绘制图形的地方，下面我们介绍另一种渲染平面。

创建隐藏渲染区域：EGL　Pbuffers

除了能够绘制可显示的窗口，还可以绘制不能显示的窗口平面，称为 pbuffers（像素缓冲区）。Pbuffers 能充分利用硬件，加速 OpenGL ES 2.0。pbuffers 常用来产生贴图纹理，如果你想使用贴图，你可以使用帧缓冲区（第 12 章帧缓冲区）来代替 pbuffers，这时效率会更高。但 pbuffers 在帧缓冲区不能使用时，也是非常有用的，比如渲染隐藏平面、在 OpenVG 中作为贴图。

创建 pbuffers 非常简单，有一点不同。也是使用 EGLConfig，但修改 EGL_SURFACE_TYPE 值包括 EGL_PBUFFER_BIT，有了合适的 EGLConfig 后，使用下面的函数创建 pbuffers。

```
EGLSurface eglCreatePbufferSurface(EGLDisplay display,
                                   EGLConfig  config, const EGLint *attribList);
```

窗口创建后，我们注意力放到我们选择的 EGLConfig 属性，操作系统显示管理。

它产生的属性列表为表 3-4

表 3-4 EGL 像素缓冲区 pbuffer 属性

Token	DescriptionDefault	Value
EGL_WIDTH	指定希望 pbuffer 宽度(单位 pixels)	0
EGL_HEIGHT	指定希望 pbuffer 高度(单位 pixels)	0
EGL_LARGEST_PBUFFER	选择最大可用的 pbuffer ，如果要求的尺寸不可用。值是 EGL_TRUE 或 EGL_FALSE.	EGL_FALSE
EGL_TEXTURE_FORMAT	指定贴图格式类型(see Chapter 9) 如果 pbuffer 被绑定到贴图。有效值是 EGL_TEXTURE_RGB,map、EGL_TEXTURE_RGBA, 和 EGL_NO_TEXTURE (这表明 pbuffer 不能被贴图直接使用).	EGL_NO_TEXTU
EGL_TEXTURE_TARGET	如果使用贴图匹配(see Chapter 9)，pbuffer 应该被联系到指定的关联贴图目标，有效值是 EGL_TEXTURE_2D 或 EGL_NO_TEXTURE.	EGL_NO_TEXTU
EGL_MIPMAP_TEXTURE	指定存储多级纹理(see Chapter 9) 是否应该被额外分配，有效值是 EGL_TRUE and EGL_FALSE.	EGL_FALSE

如果创建 eglCreatePbufferSurface 失败，这时函数调用返回 EGL_NO_SURFACE，我们可以使用 eglGetError 查询错误发生原因，可能的原因如表 3-5：

表 3-5 可能的原因创建 eglCreatePbufferSurface 失败

Error	CodeDescription
EGL_BAD_ALLOC	pbuffer 因为缺少资源而不能分配
EGL_BAD_CONFIG	提供的 EGLConfig 不是系统支持的有效 EGLConfig
EGL_BAD_PARAMETER	属性列表提供的 EGL_WIDTH 宽度和高度 EGL_HEIGHT 是负值
EGL_BAD_MATCH	下面的任何一个事件发生将产生这种错误：EGLConfig 配置不支持 pbuffer 窗口， pbuffer 被用于贴图匹配，(EGL_TEXTURE_FORMAT 不是 EGL_NO_TEXTURE), 指定的 EGL_WIDTH 和 EGL_HEIGHT 不是有效的贴图尺寸，

	EGL_TEXTURE_FORMAT 或者 EGL_TEXTURE_TARGET 中一个是 EGL_NO_TEXTURE, 但另一个的属性不是 EGL_NO_TEXTURE。
EGL_BAD_ATTRIBUTE	EGL_TEXTURE_FORMAT,EGL_TEXTURE_TARGET, 或 EGL_MIPMAP_TEXTURE 被指定，但 EGLConfig 不支持 OpenGL ES 渲染(例如仅仅 支持 OpenVG 渲染).

综上所述，例 3-5 我们创建 pbuffer。

例 3-5 创建 EGL pbuffer

```
EGLint attribList[] =
{
    EGL_SURFACE_TYPE, EGL_PBUFFER_BIT,
    EGL_RENDERABLE_TYPE, EGL_OPENGL_ES2_BIT,
    EGL_RED_SIZE, 5,
    EGL_GREEN_SIZE, 6,
    EGL_BLUE_SIZE, 5,
    EGL_DEPTH_SIZE, 1,
    EGL_NONE
};
const EGLint MaxConfigs = 10;
EGLConfig   configs[MaxConfigs];   // We'll only accept 10 configs
EGLint   numConfigs;
if(!eglChooseConfig(dpy, attribList, configs, MaxConfigs,
    &numConfigs))
{
    // Something didn't work   handle error situatio
}
else
{
    // We've found a pbuffer-capable EGLConfig
}
// Proceed to create a 512 x 512 pbuffer (or the largest available)
EGLRenderSurface   pbuffer;
EGLint attribList[] =
{
    EGL_WIDTH, 512,
    EGL_HEIGHT, 512,
    EGL_LARGEST_PBUFFER, EGL_TRUE,
    EGL_NONE
);
pbuffer = eglCreatePbufferSurface(dpy, config, attribList);
pbuffer = eglCreatePbufferSurface(dpy, config, attribList);
if(pbuffer == EGL_NO_SURFACE)
{
    switch(eglGetError())
```

```
    {
        case EGL_BAD_ALLOC:
            // Not enough resources available. Handle and recover
            break;
        case EGL_BAD_CONFIG:
            // Verify that provided EGLConfig is valid
            break;
        case EGL_BAD_PARAMETER:
            // Verify that the EGL_WIDTH and EGL_HEIGHT are
            // non-negative values
            break;
        case EGL_BAD_MATCH:
            // Check window and EGLConfig attributes to determine
            // compatibility and pbuffer-texture parameters
            break;

    }
}
// Check to see what size pbuffer we were allocated
EGLint    width;
EGLint    height;
if(!eglQuerySurface(dpy, pbuffer, EGL_WIDTH, &width) ||
    !eglQuerySurface(dpy, pbuffer, EGL_HEIGHT, &height))
{
    // Unable to query surface information.
}
```

Pbuffers 和视窗一样支持所有的 OpenGL ES 2.0 渲染设备，最主要的区别—除了你在屏幕上不能显示 Pbuffers 的内容，当你完成渲染后，不是交换缓冲区，而是从 Pbuffers 中拷贝值到你的应用程序中或者绑定 Pbuffers 作为贴图。

创建渲染环境（上下文）

渲染环境指 OpenGL ES 2.0 的包含所有项目运行需要的设置的数据结构。例如它包含着顶点、片段着色器、顶点数据矩阵，像第 2 章例子使用的一样。OpenGL ES 2.0 绘图前需要一个可用的 context。

创建 context，使用

```
EGLContext eglCreateContext(EGLDisplay display, EGLConfig config,
                            EGLContext shareContext,
                            const EGLint* attribList);
```

再次你需要用 EGLConfig 建立显示连接你的应用程序。第 3 个参数 shareContext 允许建立 EGLContexts 到各种类型数据的链接，像着色器、贴图。Contexts 的资源在第 13 章 OpenGL ES 2.0 先进的编程中讨论。我们可以输入 EGL_NO_CONTEXT 作为 shareContext 的值，说明我们不和其他 contexts 分享资源。

最后像其他 EGL 函数一样 eglCreateContext 指定一系列的属性，本例中接收 EGL_CONTEXT_CLIENT_VERSION 属性，表 3-6 说明这些属性含义。

表 3-6 使用 eglCreateContext 创建的 Context 属性

Token	DescriptionDefault	Value
EGL_CONTEXT_CLIENT_VERSION	context 类型描述和你使用的 OpenGL ES 版本有关	1 (说明是 OpenGL ES 1.X 版的 context)

我们使用 OpenGL ES 2.0 时，必要指定这个属性正确的值。

当创建 eglCreateContext 成功后，它返回一个新创建的 context 的句柄。如果 context 不能被创建，eglCreateContext 返回 EGL_NO_CONTEXT。可用 eglGetError 查询失败原因，依据我们现在所知的唯一的原因是 EGLConfig 不是有效的，返回的错误码是 EGL_BAD_CONFIG。

例 3-6 显示怎样使用一个合适的 EGLConfig 创建 context。

例 3-6 创建 EGL context

```
const ELGint attribList[] = {
    EGL_CONTEXT_CLIENT_VERSION, 2,
    EGL_NONE
};
EGLContext    context;
context = eglCreateContext(dpy, config, EGL_NO_CONTEXT, attribList);
if(context == EGL_NO_CONTEXT)
{
    EGLError error = eglGetError();
    if(error == EGL_BAD_CONFIG)
    {
        // Handle error and recover
    }
}
```

别的错误也可能被 eglCreateContext 产生，但目前，我们仅仅检查 EGLConfig 错误。成功创建 EGLContext 后，渲染前还需最后一步。

确定当前的 EGLContext

一个应用可能创建多种 EGLContext，我们需要确定联系到一个特殊的 EGLContext，渲染使用的 EGLContext，也叫做 make current。

联系这个特殊的、有 EGLSurface 的 EGLContext 使用

```
EGLBoolean eglmakeCurrent(EGLDisplay display, EGLSurface draw,
                          EGLSurface read, EGLContext context);
```

你可能主意到这个函数使用了两个 EGLSurfaces，这也是允许的，但我们设定这两个值为相同的值，即先前创建的窗口。

联系所有的 EGL 知识

我们现在总结一下本章，显示一个完整的例子，初始化 EGL，绑定一个 EGLContext 到 EGLRenderSurface。假定窗口被创建，没有任何错误，应用结束了。

例 3-7 是非常简单的和第 2 章的例子做的一样，除了分离窗口和 context，后面再解释。

例 3-7 完整的创建 EGL 窗口的过程

```
EGLBoolean initializeWindow(EGLNativeWindow nativeWindow)
{
    const EGLint    configAttribs[] =
```

```
    {
        EGL_RENDER_TYPE, EGL_WINDOW_BIT,
        EGL_RED_SIZE, 8,
        EGL_GREEN_SIZE, 8,
        EGL_BLUE_SIZE, 8,
        EGL_DEPTH_SIZE, 24,
        EGL_NONE
    };

    const EGLint    contextAttribs[] =
    {
        EGL_CONTEXT_CLIENT_VERSION, 2,
        EGL_NONE
    };
    EGLDisplay dpy;
    dpy = eglGetNativeDispay(EGL_DEFAULT_DISPLAY);
    if(dpy == EGL_NO_DISPLAY)
    {
        return EGL_FALSE;
    }
    EGLint major, minor;
    if(!eglInitialize(dpy, &major, &minor))
    {
        return EGL_FALSE;
    }
EGLConfig    config;
EGLint    numConfigs;
if(!eglChooseConfig(dpy, configAttribs, &config, 1,
    &numConfigs)) {
        return EGL_FALSE;
}
EGLSurface window;
window = eglCreateWindowSurface(dpy, config, nativeWindow, NULL);
if(window == EGL_NO_SURFACE)
{
    return EGL_FALSE;
}
EGLContext context;
context = eglCreateContext(dpy, config, EGL_NO_CONTEXT,
                                        contextAttribs);
if(context == EGL_NO_CONTEXT)
{
    return EGL_FALSE;
}
```

```
    if(!eglMakeCurrent(dpy, window, window, context))
    {
        return EGL_FALSE;
    }
    return EGL_TRUE;
}
```

例 3-8 显示打开一个 512×512 的窗口。

例 3-8 使用 Esutil 库创建窗口

```
ESContext    esContext;
const char* title = "OpenGL ES Application Window Title";
if(esCreateWindow(&esContext, title, 512, 512,
                          ES_WINDOW_RGB | ES_WINDOW_DEPTH))
{
    // Window creation failed
}
```

最后 esCreateWindow 的参数指出我们窗口的特征：

ES_WINDOW_RGB—指定颜色缓冲区属性为 RGB-based

ES_WINDOW_ALPHA—分配一个 目标透明度缓冲区

ES_WINDOW_DEPTH—分配一个深度缓冲区.

ES_WINDOW_STENCIL—分配一个模板缓冲区

ES_WINDOW_MULTISAMPLE—分配一个多重采样 缓冲区

这些窗口属性值将用来填充 EGLConfig 的属性列表（例如在先前例子中提到的 configAttribs）

渲染同步

你发现你需要在一个窗口协调多个绘图 API 渲染，你可能发现使用 OpenVG 或者操作系统窗口的字渲染比 OpenGL ES 2.0 更合适。这种情况下，你需要你的应用程序允许各种库函数分享同一个窗口。EGL 的一些函数会帮助你同步这些任务。

如果你的应用程序仅仅使用 OpenGL ES 2.0 去渲染，那么你使用 glFinish 就可以同步所有的渲染。

若非如此，比如使用了 Khronos 工作组的 OpenVG，你不知道那个 API 被使用，在交换操作系统的渲染 API 前你可以使用：

EGLBoolean eglWaitClient()

延迟执行客户程序，直到所有的 Khronos 工作组 API 渲染完毕。成功返回 EGL_TRUE。失败返回 EGL_FALSE，设定错误码为 EGL_BAD_CURRENT_SURFACE。

这个操作和 glFinish 是非常相似的，但后者不管 Khronos API 是否在运行。

如果要保证操作系统的渲染完成可以使用：

EGLBoolean eglWaitNative(EGLint engine)

指出渲染要等待先前的渲染完成，总是使用 EGL_CORE_NATIVE_ENGINE 参数，被大多数的厂商驱动支持，还有一些厂商的驱动扩展支持，成功返回 EGL_TRUE。失败返回 EGL_FALSE，设定错误码为 EGL_BAD_PARAMETER

4 着色器和编程

第 2 章，我们给出一个“Hello, Triangle: An OpenGL ES 2.0 Example,”介绍一个绘制三角形的项目，例子中我们创建两个着色器（顶点和片段）和一个项目去渲染了一个三角形。着色器和项目对象是 OpenGL ES 2.0 工作的基础，本章我们介绍全部的细节：创建着色器、编译、链接。现在我们要讲这些：

着色器和项目对象预览

创建、编译着色器

创建、链接项目

获取、设置 uniforms

获取、设置属性

着色器编译器和着色器二进制码

着色器和项目

对着色器来说，使用着色器渲染有两个基本的对象类型你需要去创建：着色器对象和项目对象。如何理解着色器对象和项目对象，可以对比 C 语言的编译器和连接器。C 语言编译器可以从项目源码产生目标码（.obj 或者 .o 文件），然后 C 连接器链接目标码成为最后的程序。

OpenGL ES 中过程是类似的。着色器对象是包含单个着色器的物体。源码输入着色器对象，着色器对象被编辑为目标格式（.obj 文件）。完成后着色器对象能够链接到项目对象上，一个项目可以有多个着色器。OpenGL ES 中一个项目中有一个顶点着色器和一个片段着色器（不能多不能少），然后链接成可执行文件，最后能用来渲染。

为产生链接着色器目标，首先要创建顶点着色器和片段着色器，编译源码创建项目，然后编译链接。如果没有错误，你能够让 GL 画出你想画的任何物体。下面告诉你执行这个过程你需要使用的 API 细节。

创建和编译着色器

第 1 步是创建着色器：

GLuint glCreateShader(GLenum type)

Type 创建着色器的类型 GL_VERTEX_SHADER 或者 GL_FRAGMENT_SHADER

使用 glCreateShader 你能依据输入参数创建一个顶点着色器或片段着色器，返回值是个新着色器的句柄，不使用了，用 glDeleteShader 删除它。

void glDeleteShader(GLuint shader)

shader 要删除的着色器句柄

主要如果着色器正连接着一个项目对象，glDeleteShader 不会立刻删除着色器，而是设置删除标记，一旦着色器不再连接项目对象，才删除着色器使用的内存。

创建着色器后，一般第 2 步你要使用 glShaderSource 提供着色器源码。

void glShaderSource(GLuint shader, GLsizei count,
const char** string,
const GLint* length)

shader	着色器对象句柄
count	着色器源码的字符数，着色器由字符串组成，每个着色器仅有一个主函数 main
string	指向着色器源码字符串矩阵的指针
length	指向整数矩阵的指针，指示每个着色器字符串长度。如果长度为 NULL，表示没有这个着色器，如果非空，是每个相应的着色器字符串长度数。如果 length 小于 0，也说明字符串为空。

指定着色器源后，下一个步骤是编译。这里要注意，不是所有的 OpenGL ES 2.0 工具提

供编译着色器的功能，（有的要求离线编译）这在后面着色器二进制码中介绍。现在假设可以在线编译，后面讨论着色器二进制码。

可以使用 glCompileShader 编译。

void glCompileShader(GLuint shader)

shader 着色器句柄

调用 glCompileShader，编译着色器对象。像其他语言一样，编译后你想知道是否产生错误，可以使用 glGetShaderiv 查询。

void glGetShaderiv(GLuint shader, GLenum pname,GLint *params)

shader 着色器句柄

pname 需要查询的信息

GL_COMPILE_STATUS

GL_DELETE_STATUS

GL_INFO_LOG_LENGTH

GL_SHADER_SOURCE_LENGTH

GL_SHADER_TYPE

Params 返回查询结果的整型指针

使用 GL_COMPILE_STATUS 参数查询编译是否成功，成功返回 GL_TRUE，失败返回 GL_FALSE。如失败，编译错误被写入 info 日志。info 日志记录了编译器的任何错误和警告。即使编译成功，同样也有 info 日志。可以使用 GL_INFO_LOG_LENGTH 查询 info 日志长度。可以调用 glGetShaderInfoLog 调用 info 日志（后面讲述）。使用 GL_SHADER_TYPE 查询着色器类型是 GL_VERTEX_SHADER 或者 GL_FRAGMENT_SHADER 。使用 GL_SHADER_SOURCE_LENGTH 查询着色器源码长度，即使着色器为空。使用 GL_DELETE_STATUS 查询着色器是否被 glDeleteShader 标记删除。

要使用 glGetShaderInfoLog 调用 info 日志，首先要查询 GL_INFO_LOG_LENGTH，分配足够的字符串用来存储 info log。

void glGetShaderInfoLog(GLuint shader, GLsizei maxLength,
GLsizei *length, GLchar *infoLog)

shader handle to the shader object to get the info log for

maxLength the size of the buffer to store the info log in

length the length of the info log written (minus the null terminator). If the length does not need to be known, this parameter can be NULL

infoLog pointer to the character buffer to store the info log in

日志没有强制的格式或必须的信息，但大多数的编译器能够返回源码错误的行数，还有的编译器提供警告和附加信息。

讲了创建着色器需要的使用行为，下面看一个例子：

例 4-1 装载着色器

```
GLuint LoadShader(GLenum type, const char *shaderSrc)
{
    GLuint shader;
    GLint compiled;

    // Create the shader object
```

```
    shader = glCreateShader(type);
    if(shader == 0)
    return 0;
    // Load the shader source
    glShaderSource(shader, 1, &shaderSrc, NULL);
        // Compile the shader
    glCompileShader(shader);
    // Check the compile status
    glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
    if(!compiled)
    {
        GLint infoLen = 0;
        glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);

        if(infoLen > 1)
        {
            char* infoLog = malloc(sizeof(char) * infoLen);
            glGetShaderInfoLog(shader, infoLen, NULL, infoLog);
            esLogMessage("Error compiling shader:\n%s\n", infoLog);

            free(infoLog);
        }
        glDeleteShader(shader);
        return 0;
    }
    return shader;
```

创建和链接项目

项目是这样一个对象，你创建的着色器要链接的对象及链接后可执行的程序。项目调用着色器是很简单的。创建项目使用 glCreateProgram。

GLuint glCreateProgram(void)

这个函数没输入参数，它返回新建项目的句柄。删除一个项目使用 glDeleteProgram。

void glDeleteProgram(GLuint program)

program 要删除项目的句柄

可以使用 glAttachShader 连接一个顶点着色器和片段着色器。

void glAttachShader(GLuint program, GLuint shader)

program 项目的句柄

shader 着色器句柄

注意，着色器能被在任何时候被连接，不需要被编译甚至不需要有源码。唯一的要求是一个项目只能有一个顶点着色器和一个片段着色器。也可以使用 glDetachShader 分离一个着色器。

void glDetachShader(GLuint program, GLuint shader)

program 项目的句柄

shader 着色器句柄

着色器连接后（被成功编译）需要链接着色器，使用 glLinkProgram：

void glLinkProgram(GLuint program)
program 项目的句柄
链接产生最后的可执行程序，链接器将检查是否链接成功。我们谈到一些条件，但详细说明将是困难的，链接器将保证每个片段着色器的变量都是顶点着色器产生的（类型相同）；链接器也保证所有的片段着色器 uniforms（常量）和顶点着色器师匹配的；链接器也保证最终的项目满足编程工具的要求（属性数、常量、变量、指令执行），典型的链接器完成后的项目能够在硬件执行。
链接后使用 glGetProgramiv 检查链接是否成功：
void glGetProgramiv(GLuint program, GLenum pname, GLint *params)
program 得到项目信息的句柄
pname 查询那种信息的参数
GL_ACTIVE_ATTRIBUTES
GL_ACTIVE_ATTRIBUTE_MAX_LENGTH
GL_ACTIVE_UNIFORMS
GL_ACTIVE_UNIFORM_MAX_LENGTH
GL_ATTACHED_SHADERS
GL_DELETE_STATUS
GL_INFO_LOG_LENGTH
GL_LINK_STATUS
GL_VALIDATE_STATUS
params 保存查询结果的整形指针
链接成功后，准备渲染。要查询项目是否有效，有一些原因会导致项目不能执行，例如：应用没有对采样器绑定一个有效的贴图。这在连接时不能发现，只能在绘制时查询。检查项目当前的执行状态使用 glValidateProgram：
void glValidateProgram(GLuint program)
program 生效的项目句柄
使用 GL_VALIDATE_STATUS 检查先前绘制是否是有效的结果。
注意：如果你使用 glValidateProgram 仅仅为了调试，这将是缓慢的，不能在渲染前随时使用。如果你的应用成功渲染，就不要使用它了。这里只要知道这个函数确实存在。
我们已经讲了创建项目，链接着色器、编译、链接获取日志，还有一些事情需要做，即设置项目为实际的渲染目标，使用 glUseProgram 函数：
void glUseProgram(GLuint program)
program 需要激活的项目句柄
我们的项目要激活了，准备渲染吧，例 4-2 是第 2 章使用过的代码：
例 4-2 创建、连接着色器、链接项目

```
// Create the program object
programObject = glCreateProgram();

if(programObject == 0)
    return 0;
glAttachShader(programObject, vertexShader);
glAttachShader(programObject, fragmentShader);
// Link the program
glLinkProgram(programObject);
```

```
    // Check the link status
    glGetProgramiv(programObject, GL_LINK_STATUS, &linked);
    if(!linked)
    {
        GLint infoLen = 0;
        glGetProgramiv(programObject, GL_INFO_LOG_LENGTH, &infoLen);

        if(infoLen > 1)
        {
            char* infoLog = malloc(sizeof(char) * infoLen);
            glGetProgramInfoLog(programObject, infoLen, NULL, infoLog);
            esLogMessage("Error linking program:\n%s\n", infoLog);

            free(infoLog);
        }
        glDeleteProgram(programObject);
        return FALSE;
    }
    // …
    // Use the program object
    glUseProgram(userData->programObject);
```

常量（Uniforms）和属性

链接项目对象时，你有一些查询需要去做。首先是找到项目使用的 Uniforms，输入着色器的只读变量，细节在下一章着色器语言讲述，这些常量集也被项目对象分享。项目对象里有一系列的常量集合，如果常量被顶点着色器和片段着色器共同使用，它们的值在两个着色器里应该是相同的。链接阶段，链接器将分配常量在项目里的实际地址，那个地址是被应用程序使用和装载的标志。

获取和使用 Uniforms

在项目里查询 Uniforms，可以使用 GL_ACTIVE_UNIFORMS 参数调用 glGetProgramiv，这将告诉你项目里实际的常量数目，如果常量处于 active，它正在被项目使用。另外你可以声明一些常量，但不使用它，链接器将会优化掉这些常量，不将它存储到实际使用的常量列表中。你能使用 GL_ACTIVE_UNIFORM_MAX_LENGTH 为参数调用 glGetProgramiv，找出在项目里最大的常量列表名字（甚至是空的终端）。

一旦我们知道实际使用的常量数目和描述的数目，我们可以使用 glGetActiveUniform 查找常量的细节：

```
void    glGetActiveUniform(GLuint program, GLuint index,
                           GLsizei bufSize, GLsizei* length,
                           GLint* size, GLenum* type,
                           char* name)
```

program	项目句柄
index	被查询的常量索引
bufSize	存储名字特征的矩阵数目
length	如果非空，将写入名字特征的矩阵数目 (less the null terminator)
size	如果 uniform 被查询出是矩阵，这个值是被项目使用的最大的矩阵元素的数目

（加 1），如果 uniform 被查询出不是矩阵，这个值是 1。

type　　要写的 uniform 类型，可以是
GL_FLOAT, GL_FLOAT_VEC2, GL_FLOAT_VEC3, GL_FLOAT_VEC4
GL_INT, GL_INT_VEC2, GL_INT_VEC3, GL_INT_VEC4, GL_BOOL
GL_BOOL_VEC2, GL_BOOL_VEC3, GL_BOOL_VEC4,
GL_FLOAT_MAT2, GL_FLOAT_MAT3, GL_FLOAT_MAT4,
GL_SAMPLER_2D, GL_SAMPLER_CUBE

name　　will be written with the name of the uniform up to bufSize number of characters. This will be a null terminated string

使用 glGetActiveUniform，你能知道所有的 uniform 的属性，你能通过 uniform 的类型确认它的名字，另外你能找出来那个常量是个矩阵，这个矩阵里最大的元素。去查找常量必须知道常量的地址，名字、类型、大小，并计算出怎么装载。有了常量的名字，可以使用 glGetUniformLocation 找到它的地址，这个地址是整形值，这个地址能被后面的函数使用（例如 glUniform1f）。

```
GLint    glGetUniformLocation(GLuint program,
                              const char* name)
```

program　　项目对象的句柄

name　　用来得到地址的常量的名字

函数返回常量地址，如果常量值在项目中未使用，将返回-1，一旦我们有常量地址以及它的类型和矩阵尺寸，我们可以使用数值装载常量。使用不同函数装载不同常量。

```
void    glUniform1f(GLint location, GLfloat x)
void    glUniform1fv(GLint location, GLsizei count,
                     const GLfloat* v)
void    glUniform1i(GLint location, GLint x)
void    glUniform1iv(GLint location, GLsizei count,
                     const GLint* v)
void    glUniform2f(GLint location, GLfloat x, GLfloat y)
void    glUniform2fv(GLint location, GLsizei count,
                     const GLfloat* v)
void    glUniform2i(GLint location, GLint x, GLint y)
void    glUniform2iv(GLint location, GLsizei count,
                     const GLint* v)
void    glUniform3f(GLint location, GLfloat x, GLfloat y,
                    GLfloat z)
void    glUniform3fv(GLint location, GLsizei count,
                     const GLfloat* v)
void    glUniform3i(GLint location, GLint x, GLint y,
                    GLint z)
void    glUniform3iv(GLint location, GLsizei count,
                     const GLint* v)
void    glUniform4f(GLint location, GLfloat x, GLfloat y,
                    GLfloat z, GLfloat w);
void    glUniform4fv(GLint location, GLsizei count,
```

```
                                const GLfloat* v)
void    glUniform4i(GLint location, GLint x, GLint y,
                    GLint z, GLint w)
void    glUniform4iv(GLint location, GLsizei count,
                                const GLint* v)
void    glUniformMatrix2fv(GLint location, GLsizei count,
                                GLboolean transpose,
                                const GLfloat* value)
void    glUniformMatrix3fv(GLint location, GLsizei count,
                                GLboolean transpose,
                                const GLfloat* value)
void    glUniformMatrix4fv(GLint location, GLsizei count,
                                GLboolean transpose,
                                const GLfloat* value)
```

location	装载常量数值的地址
count	装载矩阵元素数目的指针，只有一个常量元素时为 1，为矩阵时是 glGetActiveUniform 返回的值
transpose	这个矩阵变换参数在 OpenGL ES 2.0 必须为 FALSE。 为了保持和桌面版 OpenGL 兼容，在 OpenGL ES 2.0 不使用

装载常量函数大部分是自动完成的。使用哪个函数装载常量取决于 glGetActiveUniform 函数返回的类型。例如类型是 GL_FLOAT_VEC4，glUniform4f 或 glUniform4fv 被使用。如果返回的类型大于一个，glUniform4fv 将被使用一次装载整个矩阵。如果常量不是矩阵，glUniform4f 或 glUniform4fv 被使用。

一个值得注意的地方是 glUniform*调用不使用项目对象句柄做参数。原因是因为 glUniform*调用总是在当前的项目中绑定 glUseProgram 执行。常量值在项目对象中总是保持一致。也就是说，一旦你在项目里设定一个常量值，这个值将保持不变，甚至你激活另一个项目。在那种情况下，我们称常量值是局部的。

例 4-3 显示在项目对象里用我们描述的函数查询常量信息

例 4-3 查询激活的常量

```
GLint maxUniformLen;
GLint numUniforms;
char *uniformName;
GLint index;

glGetProgramiv(progObj, GL_ACTIVE_UNIFORMS, &numUniforms);
glGetProgramiv(progObj, GL_ACTIVE_UNIFORM_MAX_LENGTH,
                        &maxUniformLen);
uniformName = malloc(sizeof(char) * maxUniformLen);
for(index = 0; index < numUniforms; index++)
{
    GLint size;
    GLenum type;
    GLint location;
```

```
    // Get the Uniform Info
    glGetActiveUniform(progObj, index, maxUniformLen, NULL,
                                    &size, &type, uniformName);
    // Get the uniform location
    location = glGetUniformLocation(progObj, uniformName);

    switch(type)
    {
    case GL_FLOAT:
        // ...
        break;
    case GL_FLOAT_VEC2:
        // ...
        break;
          case GL_FLOAT_VEC3:
        // ...
        break;
    case GL_FLOAT_VEC4:
        // ...
        break;

    case GL_INT:
        // ...
        break;
    // ... Check for all the types ...

    default:
        // Unknown type
        break;
    }
}
```

获取和设置属性

除了查询常量信息，你还需要设置顶点属性。这和查询常量是类似的，你能使用 GL_ACTIVE_ATTRIBUTES 查询到一个属性列表，你能使用 glGetActiveAttrib 查询一个属性的内容。那是一个设置顶点矩阵，装载顶点属性值的集合。

然而设置顶点属性还需要一些有关基元和顶点着色器的知识。这在第 6 章顶点属性、顶点矩阵、缓冲区对象描述，或者你可以到第 6 章顶点着色器的顶点属性变量说明章节阅读。

着色器编程和着色器二进制码

像前面所述，我们使用的 OpenGL ES 2.0 工具支持在线源码编译，这可能不总是这样的。OpenGL ES 2.0 设计的一个目标是在硬件有限的内存上执行应用。因此，写一个着色器编译器能够用更少的代码和内存。

当你告诉 OpenGL ES 去编译和链接一个着色器时，花一点时间考虑一下编译工具必须做的事。编译器典型的解析成媒介层，像大多数的语言（例如一个抽象语法树），然后编译

这些抽象层变成机器硬件可执行的代码。理想情况下，这些编译将做一些优化，像死码移除、常量压缩等。换来节省 CPU 时间和内存的好处。

不是所有的 OpenGL ES 2.0 工具提供这样的编译器，也允许编译工具只支持二进制码。方法是 OpenGL ES 2.0 厂商提供一个离线工具能编译着色器源码成二进制格式着色器，能够被编译工具支持。没有标准的二进制格式，因此厂商能够有自己的工具。很明显这不利于移植，但也意味着厂商能够发展一个简单的编程工具。第 15 章硬件平台的 OpenGL ES and EGL，我们讨论一些二进制着色器和它们如何产生。现在我们仅介绍 API 和 API 使用方法。

首先，你能使用 GL_SHADER_COMPILER 为参数调用 glGetBooleanv，检查一个一个编译工具是否支持在线着色器编辑。GL_TRUE 证明编译器支持在线着色器编程，GL_FALSE 说明仅支持二进制着色器。支持在线编程的，说明你能够像我们在例子中那样使用 glShaderSource。你完成了一个着色器应用，能使用 glReleaseShaderCompiler。这个函数提示编程工具，你已经完成了着色器编译，可以释放资源了。注意这个函数仅仅是 hint，如果你决定编译更多的着色器，可以使用 glCompileShader，工具将重新为编译器分配资源。

```
void    glReleaseShaderCompiler(void)
```

提供一个提示，让工具释放着色器编辑器使用的资源，这仅仅是个提示，一些工具会忽略这个提示。

如果我们给了一个仅支持二进制码的工具，则它必须至少支持一种二进制格式。OpenGL ES 白皮书没有授权任何的二进制格式。二进制格式是厂商定义的，例 4-4，显示你应该怎样查询着色器编译器是否可用，以及哪种二进制格式被支持。

例 4-4 查询着色器编译器是否可用

```
GLboolean shaderCompiler;
GLint numBinaryFormats;
GLint *formats;
// Determine if a shader compiler available
glGetBooleanv(GL_SHADER_COMPILER, &shaderCompiler);
// Determine binary formats available
glGetIntegerv(GL_NUM_SHADER_BINARY_FORMATS, &numBinaryFormats);
formats = malloc(sizeof(GLint) * numBinaryFormats);
glGetIntegerv(GL_SHADER_BINARY_FORMATS, formats);
// "formats" now holds the list of supported binary formats
```

注意，一些编译器工具会源码和二进制都支持，但至少支持一个。如果使用支持二进制码的工具，使用 glShaderBinary 装载它。

```
void    glShaderBinary(GLint n, const GLuint* shaders, GLenum    binaryFormat,
                       const void* binary, GLint length)
```

n	着色器矩阵里的着色器数目
shaders	着色器矩阵句柄 依据那种二进制码被支持，它将是顶点着色器对象或者是片段着色器对象，或者两者都是，这决定于厂商的着色器二进制码扩展
binary	厂商的着色器二进制码格式
binary	被离线编译器产生的二进制码数据指针
length	二进制数据位数

一些厂商要求二进制码里包含顶点着色器和片段着色器。另外一些要求它们是独立的，然后在线链接。原因是厂商提供在线链接是困难的，所以要链接好再使用。

一旦着色器源码被提供，可以在程序里使用了。你能够认为二进制着色器码就是编译器着色器源码。仅仅的不同是着色器源码允许读回源码。这样你有两个方法使用装载着色器。

依靠工具的支持进行选择。如果你的平台不支持在线编译，可能是它不需要。但对工业应用来说，发展源码编译或是二进制码编译，都很好。不同的编译器对象被装载在不同可编译状态，他们能交换使用，让你的应用程序保持相同。

5 OpenGL ES 着色器语言

像你看到的第 1 章—OpenGL ES 2.0 介绍 ，第 2 章—OpenGL ES 2.0 例子：hello，Triangl，第 4 章—着色器和编程，着色器是 OpenGL ES 2.0 API 核心的基础。每个 OpenGL ES 2.0 程序要求一个顶点着色器和一个片段着色器去渲染一个图形。着色器概念是 API 观念的中心，我们假设在开始绘图 API 函数细节前，你已经有了写着色器的基础。

本章的目标是让你懂得下面着色器语言的观念：

变量和变量类型

矢量和矩阵创建及选择

常量

结构和阵列

运算符、流控制和函数

属性、只读变量和变量

预处理和指令

只读变量和变量压缩

精度控制和不变性

第 2 章中 Hello Triangle 的例子，我们很少介绍那些观念的细节，现在我们来介绍这些细节，让你懂得如何读和写着色器。

OpenGL ES 着色器编程语言基础

读完本书，你会看到很多着色器，开发你的应用程序，你要写很多着色器。你要懂得着色器会做什么及它怎么适合管线。否则请看第 1 章，那里我们讲了管线和怎么适合顶点着色器及片段着色器。

现在让我们看看着色器的组成，你可能已经看到它的语法和 C 语言十分类似，如果你有 C 语言基础，懂这个也不会有什么困难。但他们还是有些不同，让我们从支持的数据类型开始。

变量和变量类型

计算机图形学中，转换有两种基本的数据类型：矢量和矩阵。这也是 OpenGL ES 编程语言的中心类型。表 5-1，描述了编程语言存在的标量、矢量、矩阵数据类型。

表 5-1 OpenGL ES 编程语言数据类型

Variable Class	Types	Description
Scalars	float, int, bool	标量数据类型浮点数、整形数、布尔值
Floating-point Vectors	float, vec2, vec3, vec4	浮点型矢量，1、2、3、4 维
Integer vector	int, ivec2, ivec3, ivec4	整形矢量，1、2、3、4 维
Boolean vector	int, ivec2, ivec3, ivec4	布尔矢量，1、2、3、4 维
Matrices	mat2, mat3, mat4	浮点类型矩阵 2×2,3×3,4×4

变量必须先声明，像下面一样

```
float   specularAtten;    // A floating-point-based scalar
vec4    vPosition;        // A floating-point-based 4-tuple vector
mat4    mViewProjection; // A 4 x 4 matrix variable declaration
ivec2   vOffset;          // An integer-based 2-tuple vector
```

变量可以在声明时初始化，或以后初始化，初始化是通过构造函数进行，也可做类型转换。

变量构造函数

OpenGL ES 着色器语法对类型转换是非常严格的，变量可以给相同类型变量赋值。如果要进行变量转化，有一些构造变量可以使用。你能使用构造函数给变量初始化或者进行变量转换。变量除了构造函数赋值，也能够在声明时初始化。每一个变量都要一些构造函数。

下面我们看一下标量数据如何在初始化时赋值。

```
float    myFloat = 1.0;
bool     myBool = true;
int      myInt = 0;
myFloat = float(myBool); // Convert from bool -> float
myFloat = float(myInt);    // Convert from int -> float
myBool = bool(myInt);      // Convert from int -> bool
```

同样的构造语句能够被使用于转换和初始化矢量类型。输入参数将被转化相同类型的矢量类型（浮点、整形、布尔）有两个方法给矢量变量初始化。

如果输入的是标量，标量值赋给矢量的所有参数。

如果输入是多个标量或者是矢量，从左到右设置矢量变量的参数，如果多个矢量作为参数，那么矢量里要有至少输入矢量个数的参数。

下面是一些例子：

```
vec4 myVec4 = vec4(1.0);                  // myVec4 = {1.0, 1.0, 1.0, 1.0}
vec3 myVec3 = vec3(1.0, 0.0, 0.5);        // myVec3 = {1.0, 0.0, 0.5}
vec3 temp      = vec3(myVec3);            // temp = myVec3
vec2 myVec2 = vec2(myVec3);               // myVec2 = {myVec3.x, myVec3.y}
myVec4 = vec4(myVec2, temp, 0.0);    // myVec4 = {myVec2.x, myVec2.y,
                                          //                temp, 0.0 }
```

对矩阵来说，赋值操作是很灵活的，基本的规则是：

如果提供的是标量，那么标量值赋给矩阵主对角线元素。例如 mat4(1.0)创建一个 4×4 的单位矩阵。

一个矩阵能够被几个矢量赋值，例如 mat2 被两个 vec2s 赋值。

一个矩阵能够被对个标量赋值，这时矩阵里的每个值被从左到右依次赋值。

矩阵赋值是非常灵活的，只要提供足够的输入矢量或标量，OpenGL ES 里矩阵以列序存储，赋值时，矩阵以列序赋值。里面的例子显示了这个过程。

```
mat3 myMat3 = mat3(1.0, 0.0, 0.0,   // 第一列
                   0.0, 1.0, 0.0,   // 第二列
                   0.0, 1.0, 1.0);  // 第三列
```

矢量和矩阵元素

矩阵元素能够通过两种方式获取，使用“.”运算符或者数组下标。依据被给的元素的组成，每个被给的矩阵都能使用{x, y, z, w}, {r, g, b, a},或{s, t, r, q}表示。使用三种不同的命名表是因为有三种坐标顶点、颜色和贴图。x, r, 或 s 表示矩阵里的第一个元素，不同的命名方式仅仅是为了使用方便。或者说你可以使用矩阵时混合使用矩阵命名方式，（但不能使用.xgr，只能一次使用一种命名规则）。当使用“.”时，你也可以重新排列一个矩阵。例如，

```
vec3 myVec3 = vec3(0.0, 1.0, 2.0); // myVec3 = {0.0, 1.0, 2.0}
vec3 temp;

temp = myVec3.xyz;                        // temp = {0.0, 1.0, 2.0}
temp = myVec3.xxx;                        // temp = {0.0, 0.0, 0.0}
temp = myVec3.zyx;                        // temp = {2.0, 1.0, 0.0}
```

矩阵也可以使用[]操作符，在这种下标模式[0]代表 x, [1]代表 y。一个需要注意的是，OpenGL ES 2.0 里不支持非常量下标（整型变量）。原因是因为对一些硬件，矩阵的动态索引是困难的。OpenGL ES 2.0 技术说明里不支持这种行为，但除了一种特殊的变量类型

（uniformuniform）。细节在第 8 章，顶点着色器里，讲述着色器语言限制时讲述，这里你要意识到这个问题就好了。

矩阵被认为是多个矢量组成的，例如 mat2 被考虑是两个 vec2s，mat3 是 3 个 vec3s。对矩阵，单独的列被使用列下标[]选中。下面是例子：

```
mat4 myMat4 = mat4(1.0);       // Initialize diagonal to 1.0 (identity)
vec4 col0 = myMat4[0];         // Get col0 vector out of the matrix
float m1_1 = myMat4[1][1];     // Get element at [1][1] in matrix
float m2_2 = myMat4[2].z;      // Get element at [2][2] in matrix
```

常量

也可以声明一些 uniform 作为基本类型，uniform 是一些在着色器里不改变值的量。uniform 声明时加 const 修饰，必须在声明时初始化，例子如下：

```
const float    zero = 0.0;
const float    pi = 3.14159;
const vec4     red = vec4(1.0, 0.0, 0.0, 1.0);
const mat4     identity = mat4(1.0);
```

在 C 和 C++里，变量被声明为 const，说明是只读的，不能被修改。

结构

像 C 语言一样，可以集合几种变量成为结构。在 OpenGL ES 宣布一个结构如下：

```
struct fogStruct
{
    vec4 color;
    float start;
    float end;
} fogVar;
```

这将产生新的变量类型 fogStruct，和新的变量名 fogVar。变量能够初始化使用构造函数。定义一个结构类型，定义一个结构，一个同名的构造函数也被定义，必须是一对一的。上面的结构能够被使用下面的语法初始化：

```
struct fogStruct
{
    vec4    color;
    float start;
    float end;
} fogVar;
fogVar = fogStruct(vec4(0.0, 1.0, 0.0, 0.0),          // color
                   0.5,                               // start
                   2.0);                              // end
```

结构的构造基于结构的类型，它把每个成员做输入参数。访问结构中的元素和 C 语言相同。

```
vec4 color = fogVar.color;
float start = fogVar.start;
float end = fogVar.end;
```

数组

OpenGL ES 数组语法和 C 语言非常类似，索引以 0 开始。下面是创建数组的例子：

```
float floatArray[4];
vec4 vecArray[2];
```

有两个问题需要注意,首先编译时 OpenGL ES 不允许使用变量做数组的索引，仅仅支持整型常量表达式（例外情况，顶点着色器使用 uniform 变量做索引，在第 8 章讨论）。

OpenGL ES 中不支持创建时初始化。数组元素需要一对一的初始化，而且数组不能被 const 限制，因为没有语法初始化这样一个数组。

这些限制对熟悉 C 语言者可能是奇怪的，这些限制是基于硬件对数组索引的支持。这是因为很多 GPU 仅仅支持 常量索引（不是注册），这是困难的支持任意的索引。

运算符

表 5-2 OpenGL ES 运算符：

Operator	Type Description
*	乘
/	除
+	加
-	减
++	自加 (前缀或后缀)
--	自减 (前缀或后缀)
=	赋值
+=, -=, *=, /=	算数赋值
==, !=, <, >, <=, >=	比较运算符
&&	逻辑与
^^	逻辑异或
||	逻辑或

这些运算符使用和 C 语言非常类似。但 OpenGL ES 语法有严格的语法限制，执行运算符的变量必须有相同的类型，二进制运算符(*, /, +, -)必须是浮点变量或者是整型变量。乘运算符能够在浮点、矢量、矩阵的组合中运行。例如：

```
float myFloat;
vec4  myVec4;
mat4  myMat4;
myVec4 = myVec4 * myFloat;   // Multiplies each component of myVec4
                             // by a scalar myFloat
myVec4 = myVec4 * myVec4;    // Multiplies each component of myVec4
                             // together (e.g., myVec4 ^ 2 )
myVec4 = myMat4 * myVec4;    // Does a matrix * vector multiply of
                             // myMat4 * myVec4
myMat4 = myMat4 * myMat4;    // Does a matrix * matrix multiply of
                             // myMat4 * myMat4
myMat4 = myMat4 * myFloat;   // Multiplies each matrix component by
                             // the scalar myFloat
```

比较运算符(==, !=, <, etc.)仅能够执行标量，矢量有专门的比较函数去。

函数

函数声明和 C 语言一样，函数使用前，必须定义，它的原型必须给出。使用时非常类似 C 语言。不同是参数使用上，提供特殊的变量限定词，指示变量是否能够被函数修改。那些限定词如下表：

表 5-3 OpenGL ES 着色器语言限定词

Qualifier	Description
In	（无说明时默认值）指示参数是传值使用，不能被函数修改
Inout	说明变量是通过引用使用，它的值可以被修改，函数引用后值将改变。
Out	说明变量值不是函数输入，它的值在函数返回时被修改。

一个例子被提供，说明这些限定词的使用.

```
vec4 myFunc(inout float myFloat, // inout parameter
            out vec4 myVec4,     // out parameter
            mat4 myMat4);        // in parameter (default)
```

一个函数例子给出基本散射光计算函数。

```
vec4 diffuse(vec3 normal,
             vec3 light,
             vec4 baseColor)
{
    return baseColor * dot(normal, light);
}
```

一个主意的是，OpenGL ES 函数不能递归，原因是一些编译工具执行这个函数时，这会让这个函数在线执行，最后使 GPU 产生一些问题。编程语言这种限制的目的是为了在没有堆暂和流控制的 GPU 上使能在线编译。

内联函数

前面的章节描述了一个着色器创建函数的行为。OpenGL ES 着色器语言最强大的功能之一是提供内联函数。下面的例子是，在片段着色器中计算基本反射光的着色器代码。

```
float nDotL = dot(normal , light);
float rDotV = dot(viewDir, (2.0 * normal) * nDotL C light);
float specular = specularColor * pow(rDotV, specularPower);
```

这段代码中，使用 dot 内联函数去计算两个矢量和 pow 内联函数使一个标量成为 power。在 OpenGL ES 着色器语言中，有很多内联函数数组，被使用在必须由着色器完成的各种计算任务。在附录 B 中，我们提供一个完整的 OpenGL ES 着色器编程语言内联函数的例子。现在我们让你意识到哪有很多内联函数，为更熟悉着色器，你应该尽量去熟悉内联函数。

流控制声明

控制语句语法和 C 类似，if-then-else 逻辑也被使用，例如：

```
if(color.a < 0.25)
{
    color *= color.a;
}
else
{
    color = vec4(0.0);
}
```

条件表达式的结果必须是布尔值。或者说表达式基于布尔值计算，或者计算结果是布尔值（例如比较运算）。这是 OpenGL ES 着色器语言的条件被执行的基础。

另外 if-then-else 表达式，也可以写简单的循环。OpenGL ES 着色器语言循环有各种类型的限制。这些限制是因为 OpenGL ES 2.0 硬件的限制。最简单的 OpenGL ES 循环在编译时必须有迭代计数。例如：

```
for(int i = 0; i < 3; i++)
```

```
{
    sum += i;
}
```

使用循环时应该非常小心，基本的限制有：必须有循环迭代变量，它的值必须是增加或减小使用简单的表达式(i++, i--, i+=constant, i-=constant)，停止条件必须匹配循环索引，并且是常量表达式。在循环内部不能改变迭代的值。本质上，OpenGL ES 着色器语言不支持硬件提供循环支持。这些限制使循环表达式能够被编译器展开。

OpenGL ES 着色器语言支持的循环例子如下：

```
float myArr[4];
for(int i = 0; i < 3; i++)
{
    sum += myArr[i]; // NOT ALLOWED IN OPENGL ES, CANNOT DO
                     // INDEXING WITH NONCONSTANT EXPRESSION
}
...
uniform int loopIter;
// NOT ALLOWED IN OPENGL ES, loopIter ITERATION COUNT IS NONCONSTANT
for(int i = 0; i < loopIter; i++)
{
    sum += i;
}
```

这些限制对使用 CPU 编程的人来说可能是奇怪的。但记着 OpenGL ES 是为嵌入式设备的 API，目的是在小设备和轻电力消耗的 GPU 上使用。流控制和循环将是困难的操作。那些在循环上的限制允许简单的 GPU 应用。这些限制在早期的桌面 GPU 像 ATI 的 9500 也是普遍的。它的片段着色器也不支持循环。

uniform

一种变量类型是 uniform。uniform 是 OpenGL ES 2.0 中被输入着色器的只读值。uniform 被使用存储各种着色器需要的数据，例如：转换矩阵、光照参数或者颜色。基本上各种输入着色器的常量参数像顶点和片段（但在编译时并不知道）应该是 uniform。

uniform 应该使用修饰词被声明为全局变量，例如：

```
uniform mat4 viewProjMatrix;
uniform mat4 viewMatrix;
uniform vec3 lightPosition;
```

第 4 章我们描述了一个应用装载 uniform 到着色器。主要 uniform 的空间被顶点着色器和片段着色器分享。也就是说顶点着色器和片段着色器被链接到一起进入项目，它们分享同样的 uniform。因此一个在顶点着色器中声明的 uniform，相当于在片段着色器中也声明过了。当应用程序装载 uniform 时，它的值在顶点着色器和片段着色器都可用。

另一个需要注意的是，uniform 被存储在硬件被称为常量存储，这是一种分配在硬件上的存储常量值的空间。因为这种存储需要的空间是固定的，在程序中这种 uniform 的数量是受限的。这个限制能通过读 gl_MaxVertexUniformVectors 和 gl_MaxFragmentUniformVectors 编译变量得出。（或者用 GL_MAX_VERTEX_UNIFORM_VECTORS 或 GL_MAX_FRAGMENT_UNIFORM_ VECTORS 为参数调用 glGetIntegerv）OpenGL ES 2.0 必须至少提供 128 个顶点着色器 uniform 和 16 个片段着色器 uniform。但可以更多，第 8 章顶点着色器和第 10 章片段着色器里我们讲述那些限制和变量查询。

属性

OpenGL ES 着色器语言的另一个变量是属性。属性变量仅仅在顶点着色器中被使用，逐顶点的指定顶点着色器的输入。典型的被用来储存位置、法线、贴图坐标和颜色数据。关键是懂得属性是每个顶点被绘制的详细数据。它实际上是着色器的使用者决定什么数据是属性。例 5-1 是一个简单的有位置和贴图坐标属性着色器例子。

例 5-1 简单的顶点着色器

```
uniform mat4 u_matViewProjection;
attribute vec4 a_position;
attribute vec2 a_texCoord0;
varying vec2 v_texCoord;
void main(void)
{
    gl_Position = u_matViewProjection * a_position;
    v_texCoord = a_texCoord0;
}
```

着色器的两个顶点属性是 a_position 和 a_texCoord0 被应用程序装载。基本上应用将创建一个包含位置和贴图坐标的顶点数组为每个顶点使用，我们在第 6 章解释这个过程，顶点属性、顶点数组和和缓冲区目标。现在你对顶点着色器应该知道每个顶点都有属性。

像 uniform 一样，硬件对顶点着色器的属性变量数量有限制。最大的属性数通过工具编译支持 gl_MaxVertexAttribs 确定。（或者使用 GL_MAX_VERTEX_ATTRIBS 为参数调用 glGetIntegerv 查询）最小数是 8。如果为确保你的程序能在任何 OpenGL ES 2.0 工具上编译，确保你的属性不超过 8。属性限制在第 8 章讨论。

变量

最后讨论的是变量。变量被用来存储顶点着色器的输出和片段着色器的输入。基本上每个顶点着色器把输出数据转变成一个或更多片段着色器的输入。那些变量也被片段着色器声明（类型需匹配），并且在光栅化阶段被线性插补变成基元（更多细节在第 6 章光栅化）。变量声明的例子如下：

```
varying  vec2  texCoord;
varying  vec4  color;
```

变量声明在顶点着色器和片段着色器都存在，像上面提到的，变量是顶点着色器的输出和片段着色器的输入。它们必须被相同的声明。像 uniform 和属性，变量也有数量的限制（硬件上它们被用来做插补）。工具支持的最大数目是 gl_MaxVaryingVectors（使用 GL_MAX_VARYING_VECTORS 为参数调用 glGetIntegerv 查询）。最小数目是 8.

例 5-2 是顶点着色器和片段着色器的变量如何匹配的声明。

例 5-2 顶点着色器和片段着色器匹配变量的声明

```
// Vertex shader
uniform mat4 u_matViewProjection;
attribute vec4 a_position;
attribute vec2 a_texCoord0;
varying vec2 v_texCoord; // Varying in vertex shader
void main(void)
{
    gl_Position = u_matViewProjection * a_position;
    v_texCoord = a_texCoord0;
```

```
}
// Fragment shader
precision mediump float;
varying vec2 v_texCoord; // Varying in fragment shader
uniform sampler2D s_baseMap;
uniform sampler2D s_lightMap;
void main()
{
    vec4 baseColor;
    vec4 lightColor;

    baseColor = texture2D(s_baseMap, v_texCoord);
    lightColor = texture2D(s_lightMap, v_texCoord);
    gl_FragColor = baseColor * (lightColor + 0.25);
}
```

预处理和指令

The OpenGL ES 着色器语言的预处理和标准 C++预处理一样。使用下面的方法定义宏指令和条件测试。

#define
#undef
#if
#ifdef
#ifndef
#else
#elif
#endif

主要宏不能带参数定义（C++ 可以），#if,#else, 和 #elif 能被使用测试 defined 是否定义了一个宏指令。下面是宏预定义和它们的描述

__LINE__ // 被着色器当前的行数取代
__FILE__ // 在 OpenGL ES 2.0 中总是 0
__VERSION__ // OpenGL ES 着色器语言版本 (e.g., 100)
GL_ES // 被 ES 着色器定义为值 1

#error 在着色器编译时将产生的编译错误记录到日志中，这个值被设定到 100，着色器版本应该被设定，是为了未来新的版本和新的特征。它应该在源码开始时被写入。下面显示版本的使用：

```
#version 100 // OpenGL ES Shading Language v1.00
```

另一个重要的指令是#extension，被用来设定扩展行为。供应商发展着色器语言时，他们将创造语言扩展说明书（像 GL_OES_texture_3D），着色器必须告诉编译器是否允许扩展能够被使用。下面显示了#extension 的使用：

```
// Set behavior for an extension
#extension extension_name : behavior
// Set behavior for ALL extensions
#extension all : behavior
```

扩展的名字前，一个内容提要将要求（像 GL_OES_texture_3D）。它指示扩展的行为。这有

四种，如例 5-4：

表 5-5 扩展行为

Extension Behavior	Description
Require	扩展是被要求的。如果扩展不被支持预编译将产生一个错误，如果扩展全部不被支持，将总是产生错误。
Enable	扩展被使能。如果扩展被不支持，将产生警告。如果扩展被使能语言可以被编译，如果扩展全部不被支持，将总是产生错误。
Warn	警告任何使能的扩展，除非扩展被使用其他方式指定。如果指定全部的扩展，无论是否使用扩展，都产生警告。如果扩展不支持，也产生警告。
Disable	扩展不使能，如果使用扩展将产生错误，如果指定所有的扩展（默认情况）没有扩展使能。

例如， 3D 贴图扩展不被支持，你想让预编译产生警告（如果被支持，着色器将正常进行）。你在你的着色器顶上加上：

```
#extension GL_OES_texture_3D : enable
```

常量和变量包装

前面提到，常量和变量的硬件资源是固定的。uniform 典型的被存储在常量区，被认为是物理上的矢量数组。变量典型的被插补存储，也被存储为矢量数组。你可能注意到着色器能声明 uniform 和各种类型的变量包括标量、各种矢量和矩阵。问题是这些各种变量如何匹配硬件的物理空间。换句话说，OpenGL ES 2.0 支持的 8 种矢量变量如何存储。

OpenGL ES 2.0 中有如何包装变量和 uniform 匹配物理存储空间的规则。规则基于物理存储空间被组织成在每个存储区有四列（每个矢量一列）和一行。包装规则如此复杂是为了保持代码的整洁。换句话说，包装规则不对未包装的数据重新排序而产生额外的结构。而且包装规则寻找对物理地址空间尽量优化，而不是消极的。我们看一个例子 uniform 如何被声名和包装。

```
uniform mat3 m;
uniform float f[6];
uniform vec3 v;
```

如果不封装，很多物理空间被浪费，矩阵 m 用了 3 行，矩阵 f 用了 6 行，矩阵 v 用 1 行。这将一共用掉 10 行。图 5-1 显示了不使用任何封装的情况。

图 5-2 显示了变量被优化后的封装情况。

优化和仅仅 6 行需要。你可以主要到 f 的元素跨出了边界，原因是 GPU 索引使用矢量地址索引，当封装跨出了行边界也能够被使用。

Location	X	Y	Z	W
0	M[0].x	m[0].y	m[0].z	-
1	M[1].x	m[1].y	m[1].z	-
2	M[2].x	m[2].y	m[2].z	-
3	f[0]	-	-	-
4	f[1]	-	-	-
5	f[2]	-	-	-
6	f[3]	-	-	-
7	f[4]	-	-	-
8	f[5]	-	-	-
9	v.x	v.y	v.z	-6

Figure 5-1 Uniform Storage without Packing

Location	X	Y	Z	W
0	M[0].x	m[0].y	m[0].z	f[0]
1	M[1].x	m[1].y	m[1].z	f[1]
2	M[2].x	m[2].y	m[2].z	f[2]
3	v.x	v.y	v.z	f[3]
4	-	-	-	f[4]
5	-	-	-	f[5]

Figure 5-2 Uniform Storage with Packing

所有的封装是透明的，除了一个细节，uniform 和变量的数目。如果你想让着色器程序能够运行在所有的 OpenGL ES 2.0 编程工具上，你能使用的 uniform 和变量在封装后不能超过最小值。因此懂得封装对移植 OpenGL ES 2.0 应用的好处。

质量控制

一个和桌面版本不同的新特征是质量控制的引入。质量控制是编程者能够指定着色器输出时计算执行的精度。变量能够被指定低、中和高质量。质量控制被使用指示编译器去执行变量可能的最低的范围和精度。低质量让着色器运行的更快、更有效率，当然也影响精度。如果没有使用好，会留下人工痕迹。主要 OpenGL ES 2.0 的技术说明书里没有规定多种显示质量必须被硬件支持。如果能够完美计算高质量的精度，那么忽略低质量。不论如何一些工具使用低质量可能也是先进的。

精度控制能够被使用在任何浮点和整型数变量。指定一个 lowp,mediump 或 highp 的精度控制方法如下：

highp vec4 position;
varying lowp vec4 color;
mediump float specularExp;
另外对精度控制也有默认的值，就是说没有声明精度控制，对不同类型有默认的精度控制值。默认的精度在顶点着色器和片段着色器的头部定义：
precision highp float;
precision mediump int;

对所有基于浮点的变量默认的精度是浮点值，基于整型的变量默认的精度是整型。在着色器中， 如果没有定义指定精度，默认对 int 和 float 的精度都是高。换句话说在顶点着色器中没有指定进度控制的变量将是高质量。而片段着色器规则却不同。对浮点值没有默认的精度控制。每个着色器必须声明默认的着色器浮点精度或指定每个浮点变量的精度。即 OpenGL ES 2.0 不要求片段着色器支持高精度。决定是否高精度被支持是片段着色器是否定义了 GL_FRAGMENT_PRECISION_HIGH 宏(和工具输出 OES_fragment_precision_high 扩展字符串)。

本书中，在片段着色器前面经常会看见这样的代码：

```
#ifdef GL_FRAGMENT_PRECISION_HIGH
    precision highp float;
#else
    precision mediump float;
#endif
```

这将保证着色器工具编译时支持中等和高等的精度。

最后需要指出的是精度质量控制依靠编译工具的范围和精度。在 14 章，状态查询中有 API 决定编译工具的精度和范围。

不变性

最后要讨论的题目是不变性。在 OpenGL ES 着色器编程语言里，invariant 是被用于任何顶点着色器的变量输出的关键字。它意味着什么和为什么需要他呢。着色器被编译时可能进行优化，一些指令被重新整理。指令重新整理意味着两个着色器之间平等的计算不保证产生相同的结果。这是个问题在多通道着色器特殊情况下，依稀物体使用透明混合来绘制时。如果精度被使用来计算输出位置是不准确一样，精度的不同将导致 artifacts。这在 Z fighting 情况下经常发生。每个像素很小的 Z 精度不同引起了不同的反光。

下面的例子显示了进行多通道渲染时 invariance 对得到正确的结果是非常重要的。下面的圆环被绘制用两种方式。片段着色器先计算镜面反射光，再计算环境光和散射光。顶点着色器不使用 invariance，因此小的精度不同引起了 Z 光在图 5-3 中。

Figure 5-3 Z Fighting Artifacts Due to Not Using Invariance

同样的多通道顶点着色器使用 invariance 产生正确的图像看图 5-4.

Figure 5-4 Z Fighting Avoided Using Invariance

引入 invariance 让着色器这些同样的计算被使用，输出的值也是相同的。不变应使用关键可在变量在声明时或者变量已经被什么后例如;

```
invariant gl_Position;
invariant varying texCoord;
```

一旦输出被宣布为 invariance，同样的计算输入相同，编译器保证输出结果相同。例如

你有两个顶点着色器使用多通道图像投射矩阵依据输入计算输出。你能保证那些位置是invariance。

```
uniform mat4 u_viewProjMatrix;
attribute vec4 a_vertex;
invariant gl_Position;
void main
{
    //
    gl_Position = u_viewProjMatrix * a_vertex; // Will be the same
                                               // value in all
                                               // shaders with the
                                               // same viewProjMatrix
                                               // and vertex
}
```

可以使用#pragma 指令保证全局变量保持 invariant。

```
#pragma STDGL invariant(all)
```

或者说，编译器需要保证 invariance，对优化作出限制。所以 invariance 仅仅在需要时才能被使用，它执行的可能很慢。如果必须要求全部变量 invariance，使用#pragma 去使能全局。注意 invariance 不意味着对一个 GPU 的计算有相同的结果，不意味着在任何 OpenGL ES 的编译工具上计算将是不变的。

6 顶点属性，顶点矩阵和缓冲区目标

本章描述 OpenGL ES 2.0 的顶点属性。具体指指定它们的数据格式，绑定顶点属性索引到着色器里合适的顶点属性名字。读完这一章，你应该有好的理解什么是顶点属性，怎么绘制顶点属性基元。

顶点数据，也称为顶点属性，指每一个顶点数据。指能被用来描述每个顶点的数据，或能被所有顶点使用的常量值。例如你想绘制一个具有颜色的立方体三角形（这个例子看图 6-1，颜色是黑色的）。你指定一个恒定的值将用于三角形的所有三个顶点。但无论怎么样，三角形的三个顶点位置是不同的，你需要指定一个顶点矩阵存储三个位置值。

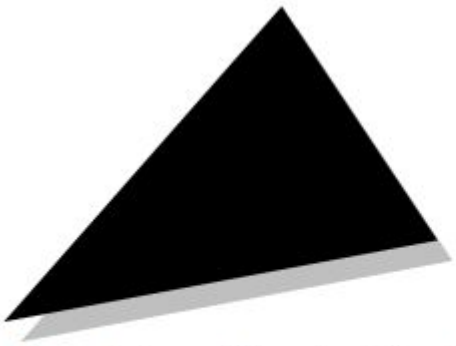


Figure 6-1 Triangle with a Constant Color Vertex and Per-Vertex Position Attributes

在 OpenGL ES 1.1 中，顶点属性有定义好的名字像：position, normal, color, 和 texture coordinates。这是容易理解的因为 OpenGL ES 1.1 使用的是固定行为的管道。OpenGL ES 2.0 使用可编程管道，开发者可以在着色器里指定自己命名的属性名字。支持自己命名的管道（像 generic）成为 OpenGL ES 2.0 的一个要求。如果自己命名的顶点属性被 API 支持，那么就不再需要预定义属性名字，因为应用能够自动匹配自己命名的顶点属性。

指定顶点属性数据

像前面提到的那样，仅仅自己命名的顶点属性被支持。每个顶点的属性数据能够被指定使用顶点矩阵或者为基元的每个顶点使用一个常量值。

OpenGL ES 2.0 必须至少支持 8 个顶点属性。应用应该能够查询编译器支持的确切属性数。下面的程序指出如何查询。

```
GLint maxVertexAttribs;     // n will be >= 8
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &maxVertexAttribs);
```

顶点属性常量

常量顶点属性是指基元的所有顶点属性是相同的，因此仅仅对基元的所有顶点仅仅需要指定一个值。

顶点属性常量使用下面的函数指定：

```
void glVertexAttrib1f(GLuint index, GLfloat x);
void glVertexAttrib2f(GLuint index, GLfloat x, GLfloat y);
void glVertexAttrib3f(GLuint index, GLfloat x, GLfloat y, GLfloat z);
void glVertexAttrib4f(GLuint index, GLfloat x, GLfloat y, GLfloat z,GLfloat w);
void glVertexAttrib1fv(GLuint index, const GLfloat *values);
```

void glVertexAttrib2fv(GLuint index, const GLfloat *values);
void glVertexAttrib3fv(GLuint index, const GLfloat *values);
void glVertexAttrib4fv(GLuint index, const GLfloat *values);

函数 glVertexAttrib* 使用 index 装载一般的顶点属性值。glVertexAttrib1f 和 glVertexAttrib1fv 装载(x, 0.0, 0.0, 1.0)到一般的顶点属性。glVertexAttrib2f 和 glVertexAttrib2fv 装载(x,y, 0.0, 1.0)。glVertexAttrib3f 和 glVertexAttrib3fv 装载(x, y, z, 1.0)。glVertexAttrib4f 和 glVertexAttrib4fv 装载(x, y, z, w)。

问题是：OpenGL 2.0 支持 byte, unsigned byte, short, unsigned short, int, unsigned int, float, 和 double。为什么 OpenGL ES 2.0 仅仅支持浮点变量？原因是常量顶点属性使用的频率低。因为不经常使用，而且存储它们不像存储单一的浮点值方便，所以 ES 2.0 仅仅支持浮点变量。

顶点数组

顶点数组指每个顶点的属性数据即存储在应用程序地址空间（OpenGL ES 叫 client space）缓冲区的数据。它们提供有效的和灵活的方法指定顶点属性数据。顶点矩阵使用 glVertexAttribPointer 函数指定：

void glVertexAttribPointer(GLuint index, GLint size,GLenum type,GLboolean normalized,
GLsizei stride,const void *ptr)

index 定点属性索引值 0 到支持的最大顶点属性－1.

size 顶点属性的矩阵数组组成元素的索引，有效值是 1—4

type 数据格式，有效值：

GL_BYTE
GL_UNSIGNED_BYTE
GL_SHORT
GL_UNSIGNED_SHORT
GL_FLOAT
GL_FIXED
GL_HALF_FLOAT_OES*

Normalized 用于指定非浮点数据格式转变成浮点型数据时是否标准化

stride 顶点属性的成分，指定相邻的数据存储间隔的空间大小。Stride 指定数据索引 I 和 I+1 的增量。如果为 0，数据连续存储，如果> 0，stride 值是数据和下一个数据增量值。

*GL_HALF_FLOAT_OES 是被 OpenGL ES 2.0 支持的可选的顶点数据格式。这个顶点数据格式是否被支持依据是否命名 GL_OES_vertex_half_float 扩展字符。想知道这个特征是否被 OpenGL ES 2.0 支持，看看是否 glGetString(GL_EXTENSIONS)是否在返回扩展列表中有 GL_OES_vertex_half_float。

我们提供几个例子解释怎么使用 glVertexAttribPointer 指定顶点属性。一般的分配和存储顶点属性数据是：

存储所有的顶点属性在一个单一的缓冲区中，这种存储顶点属性的方法叫 array of structures，这种方法描述一个顶点的所有属性，我们有每个顶点属性的一个矩阵。

存储每个顶点属性到分开的缓冲区，这种存储顶点属性的方法叫 structure of arrays.。

每个顶点有四个属性—位置、法线、两个贴图坐标，这些属性被存储在一个缓冲区中，被所有顶点分享。顶点位置属性是三个浮点数(x, y, z)的矢量。法线也是三个浮点数的矢量，每个贴图坐标是两个浮点数的矢量。图 6-2 给出了缓冲区中内存的分布。

Figure 6-2 Position, Normal, and Two Texture Coordinates Stored As an Array

例 6-1 描述了使用 glVertexAttribPointer 指定这四个矢量属性。

例 6-1 结构阵列

```
#define VERTEX_POS_SIZE                 3     // x, y and z
#define VERTEX_NORMAL_SIZE              3     // x, y and z
#define VERTEX_TEXCOORD0_SIZE           2     // s and t
#define VERTEX_TEXCOORD1_SIZE           2     // s and t
#define VERTEX_POS_INDX                 0
#define VERTEX_NORMAL_INDX              1
#define VERTEX_TEXCOORD0_INDX           2
#define VERTEX_TEXCOORD1_INDX           3
// the following 4 defines are used to determine location of various
// attributes if vertex data is are stored as an array of structures
#define VERTEX_POS_OFFSET               0
#define VERTEX_NORMAL_OFFSET            3
#define VERTEX_TEXCOORD0_OFFSET         6
#define VERTEX_TEXCOORD1_OFFSET         8
#define VERTEX_ATTRIB_SIZE      VERTEX_POS_SIZE + \
                                VERTEX_NORMAL_SIZE + \
                                VERTEX_TEXCOORD0_SIZE + \
                                VERTEX_TEXCOORD1_SIZE
float *p    = malloc(numVertices * VERTEX_ATTRIB_SIZE
                     * sizeof(float));
// position is vertex attribute 0
glVertexAttribPointer(VERTEX_POS_INDX, VERTEX_POS_SIZE,
                      GL_FLOAT, GL_FALSE,
                      VERTEX_ATTRIB_SIZE * sizeof(float),p);
// normal is vertex attribute 1
glVertexAttribPointer(VERTEX_NORMAL_INDX, VERTEX_NORMAL_SIZE,
                      GL_FLOAT, GL_FALSE,
                      VERTEX_ATTRIB_SIZE * sizeof(float),
                      (p +    VERTEX_NORMAL_OFFSET));
// texture coordinate 0 is vertex attribute 2
glVertexAttribPointer(VERTEX_TEXCOORD0_INDX, VERTEX_TEXCOORD0_SIZE,
                      GL_FLOAT, GL_FALSE,
                      VERTEX_ATTRIB_SIZE * sizeof(float),
                      (p +    VERTEX_TEXCOORD0_OFFSET));
// texture coordinate 1 is vertex attribute 3
glVertexAttribPointer(VERTEX_TEXCOORD1_INDX, VERTEX_TEXCOORD1_SIZE,
```

```
                                GL_FLOAT, GL_FALSE,
                                VERTEX_ATTRIB_SIZE * sizeof(float),
                                (p + VERTEX_TEXCOORD1_OFFSET));
```

例 6-2 描述了位置、法线、贴图坐标 0 和 1 被存储在分离的缓冲区中

例 6-2 阵列结构

```
float *position    = malloc(numVertices * VERTEX_POS_SIZE *
                            sizeof(float));
float *normal      = malloc(numVertices * VERTEX_NORMAL_SIZE *
                            sizeof(float));
float *texcoord0 = malloc(numVertices * VERTEX_TEXCOORD0_SIZE *
                            sizeof(float));
float *texcoord1 = malloc(numVertices * VERTEX_TEXCOORD1_SIZE *
                            sizeof(float));
// position is vertex attribute 0
glVertexAttribPointer(VERTEX_POS_INDX, VERTEX_POS_SIZE,
                                GL_FLOAT, GL_FALSE,
                                VERTEX_POS_SIZE * sizeof(float), position);
// normal is vertex attribute 1
glVertexAttribPointer(VERTEX_NORMAL_INDX, VERTEX_NORMAL_SIZE,
                                GL_FLOAT, GL_FALSE,
                                VERTEX_NORMAL_SIZE * sizeof(float), normal);
// texture coordinate 0 is vertex attribute 2
glVertexAttribPointer(VERTEX_TEXCOORD0_INDX, VERTEX_TEXCOORD0_SIZE,
                                GL_FLOAT, GL_FALSE, VERTEX_TEXCOORD0_SIZE *
                                sizeof(float), texcoord0);
// texture coordinate 1 is vertex attribute 3
glVertexAttribPointer(VERTEX_TEXCOORD1_INDX, VERTEX_TEXCOORD1_SIZE,
                                GL_FLOAT, GL_FALSE,
                                VERTEX_TEXCOORD1_SIZE * sizeof(float),
                                texcoord1);
```

性能提示

怎么存储不同的顶点属性

我们使用了两个不同的方法存储顶点属性：array of structures 和 structure of arrays。那种方法对硬件来说是更有效的，答案是 array of structures。原因是每个顶点的属性数据能够被连续的读出，这种内存结构更有效。但使用 array of structures 不好的是当想去修改指定的属性时。如果一个顶点属性需要被修改（像贴图坐标），这将跳跃着更新顶点缓冲区。当一个顶点缓冲区作为缓冲区对象被支持时，整个顶点属性缓冲区需要被重新装载。用它的原始结构在分离的缓冲区动态的存储顶点属性，能避免这种无效。

顶点缓冲区属性那种数据结构被使用

顶点属性数据格式被使用参数 type 调用 glVertexAttribPointer 函数指定，这样做不但影响顶点属性的绘图数据的存储要求，也影响全部的执行工作，它是渲染帧时的内存带宽的要求。数据量越小，对带宽要求越低。我们建议，如果可以使用 GL_HALF_FLOAT_OES。贴图坐标、法线、副法线、切线矢量的构成使用 GL_HALF_FLOAT_OES 存储是好的选择。每个顶点颜色的四个组成使用 GL_UNSIGNED_BYTE 存储。我们建议使用

GL_HALF_FLOAT_OES 存储顶点位置，但这是很少的情况，大多数情况下，位置使用 GL_FLOAT 或 GL_FIXED 存储。

如何标准化 glVertexAttribPointer 工作

顶点属性在被顶点着色器使用前，作为单一精度的浮点值被存储在内存中。如果顶点属性的数据类型不是浮点数，那么它们的值将在着色器使用前转变为浮点值。规一化标志指示非浮点顶点属性数据转化为单一精度的浮点值。如果归一化标准为 false，顶点数值被直接转化为浮点值，转化非浮点变量为浮点类型是相似的，下面是一个例子。

```
GLfloat     f;
GLbyte      b;
f = (GLfloat)b;    // f represents values in the range [-128.0, 127.0]
```

如果归一化为 true，顶点数据类型如果是 GL_BYTE, GL_SHORT 或 GL_FIXED 被匹配到[-1.0，1.0]，数据类型如果是 GL_UNSIGNED_BYTE or GL_UNSIGNED_SHORT 被匹配到[0.0, 1.0]。

表 6-1 描述非浮点值数据类型归一化转换过程。

表 6-1 数据转换

Vertex Data Format	Conversion to Floating Point
GL_BYTE	（2c+1）/(2^8-1)
GL_UNSIGNED_BYTE	c/(2^8-1)
GL_SHORT	（2c+1）/(2^{16}-1)
GL_UNSIGNED_SHORT	c/(2^{16}-1)
GL_FIXED	c/2^{16}-
GL_FLOAT	c
GL_HALF_FLOAT_OES	c

在常量顶点属性和顶点阵列之间选择

应用能确认 OpenGL ES 是否为顶点阵列使用常量数据或数据。图 6-3 描述了 OpenGL ES 2.0 中的这个工作。

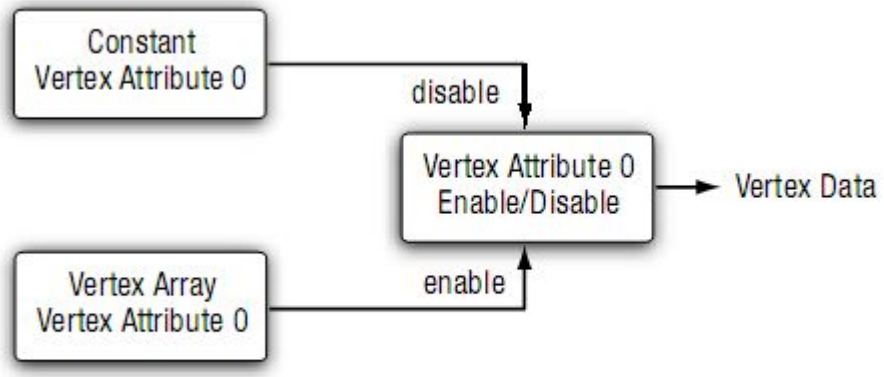


Figure 6-3 Selecting Constant or Vertex Array Vertex Attribute

glEnableVertexAttribArray 和 glDisableVertexAttribArray 被用来使能或不使能一个顶点属性阵列。如果顶点属性阵列被不使能，那么常量顶点属性数据被使用。

```
void    glEnableVertexAttribArray(GLuint index);
void    glDisableVertexAttribArray(GLuint index);
```

index 指定使用的顶点属性索引，值是 0 到最大值－1

例 6-3 描述怎样绘制一个三角形，一个使用顶点属性常量，另一个使用顶点属性阵列。

例 6-3 使用顶点属性常量，顶点属性阵列

```
GLbyte vertexShaderSrc[] =
        "attribute vec4 a_position;       \n"
        "attribute vec4 a_color;          \n"
        "varying vec4      v_color;       \n"
        "void main()                      \n"
        "{                                \n"
        "       v_color = a_color;        \n"
        "       gl_Position = a_position; \n"
        "}";
GLbyte fragmentShaderSrc[] =
        "varying vec4 v_color;            \n"
        "void main()                      \n"
        "{                                \n"
        "       gl_FragColor = v_color;   \n"
        "}";
GLfloat     color[4] = { 1.0f, 0.0f, 0.0f, 1.0f };
GLfloat     vertexPos[3 * 3];    // 3 vertices, with (x,y,z) per-vertex
GLuint       shaderObject[2];
GLuint       programObject;
shaderObject[0] = LoadShader(vertexShaderSrc, GL_VERTEX_SHADER);
shaderObject[1] = LoadShader(fragmentShaderSrc, GL_FRAGMENT_SHADER);
programObject = glCreateProgram();
glAttachShader(programObject, shaderObject[0]);
glAttachShader(programObject, shaderObject[1]);
glVertexAttrib4fv(0, color);
glVertexAttribPointer(1, 3, GL_FLOAT, GL_FALSE, 0, vertexPos);
glEnableVertexAttribArray(1);
glBindAttribLocation(programObject, 0, "a_color");
glBindAttribLocation(programObject, 1, "a_position");
glLinkProgram(programObject);
glUseProgram(programObject);
glDrawArrays(GL_TRIANGLES, 0, 3);
```

在例子中顶点属性 color 使用常量值，vertexPos 使用了顶点阵列。对三角形的颜色值是相同的，位置属性是三角形顶点变量。

在顶点着色器中声明顶点属性变量

我们看了 OpenGL ES 中怎么指定顶点属性，现在看怎么在着色器里声明顶点属性变量。

在顶点着色器中，一个变量被使用限定词 attribute 声明为顶点属性。这个限定词仅仅在顶点着色器中使用。如果被用于片段着色器，片段着色器编译时将会产生错误。

一个顶点属性声明的例子被给：

```
attribute vec4      a_position;
attribute vec2      a_texcoord;
attribute vec3      a_normal;
```

限定词 attribute 仅仅被用于数据类型 float, vec2,vec3,vec4,mat2,mat3, 和 mat4。顶点属

性不能被声明为数组或结构。下面的声明是非法的，将产生编译错误。

```
attribute foo_t    a_A;     // foo_t is a structure
attribute vec4     a_B[10];
```

OpenGL ES 2.0 应用支持 GL_MAX_VERTEX_ATTRIBS vec4 顶点属性。一个被声明为 float 或 vec2 或 vec3 的顶点属性将被认为是 vec4 属性。声明为 mat2,mat3, 或 mat4 的顶点属性将被算为 2、3 或 4 vec4 属性，是独立的，不会被封装，不像 uniform 和变量被编译器自动封装。每个成员被编译器工具作为 32 位单精度浮点值存储在内部。当声明尺寸小于 vec4 的顶点属性时，请考虑清楚，因为最大的顶点属性变量是个有限的资源，将它们组合成一个 vec4 属性代替独立声明它们将是更有效的封装。

在着色器里声明为顶点属性的变量是只读资源，不能被修改。下面的代码将产生编译错误。

```
attribute vec4     a_pos;
uniform     vec4     u_v;
void main()
{
    a_pos = u_v; <--- cannot assign to a_pos as it is read-only
}
```

一个在顶点着色器内声明的属性如果不激活不能被使用，如果着色器属性大于 GL_MAX_VERTEX_ATTRIBS，着色器将链接失败。

一旦项目成功链接，我们需要找到实际链接到着色器被使用的顶点着色器属性数目。下面的函数描述怎么得到实际顶点属性数。

```
glGetProgramiv(progam, GL_ACTIVE_ATTRIBUTES, &numActiveAttribs);
```

函数 glGetProgamiv 的细节在第 4 章，着色器和项目中描述。

实际被项目使用的顶点着色器和它们的数据类型能使用 glGetActiveAttrib 命令查询。

```
void     glGetActiveAttrib(GLuint program, GLuint index,
                           GLsizei bufsize, GLsizei *length,
                           GLint *size, GLenum *type,
                           GLchar *name)
```

program	先前成功链接的项目名字
index	指定需要查询的顶点属性，值是 0 到 GL_ACTIVE_ATTRIBUTES-1。GL_ACTIVE_ATTRIBUTES 值使用 glGetProgramiv 确定。
bufsize	指定最大的能被作为名字的数目，包括空。
Length	返回的名字长度，包括空，长度不能是空。
Type	返回的属性类型，有效值有： GL_FLOAT GL_FLOAT_VEC2 GL_FLOAT_VEC3 GL_FLOAT_VEC4 GL_FLOAT_MAT2 GL_FLOAT_MAT3 GL_FLOAT_MAT4
Size	返回属性的大小，指定返回单元类型的单位。如果不是数组，应该是 1，如果是数组，返回数组的尺寸。
Name	在顶点着色器声明的属性变量的名字

函数 glGetActiveAttrib 提供选择索引的属性的信息。像前面所诉，索引值必须是 0 和 GL_ACTIVE_ATTRIBUTES – 1 之间。GL_ACTIVE_ATTRIBUTES 值是使用 glGetProgramiv 查询。0 索引选择第 1 个激活的属性，GL_ACTIVE_ATTRIBUTES –1 选择最后一个顶点属性。

在顶点着色器绑定顶点属性到属性变量

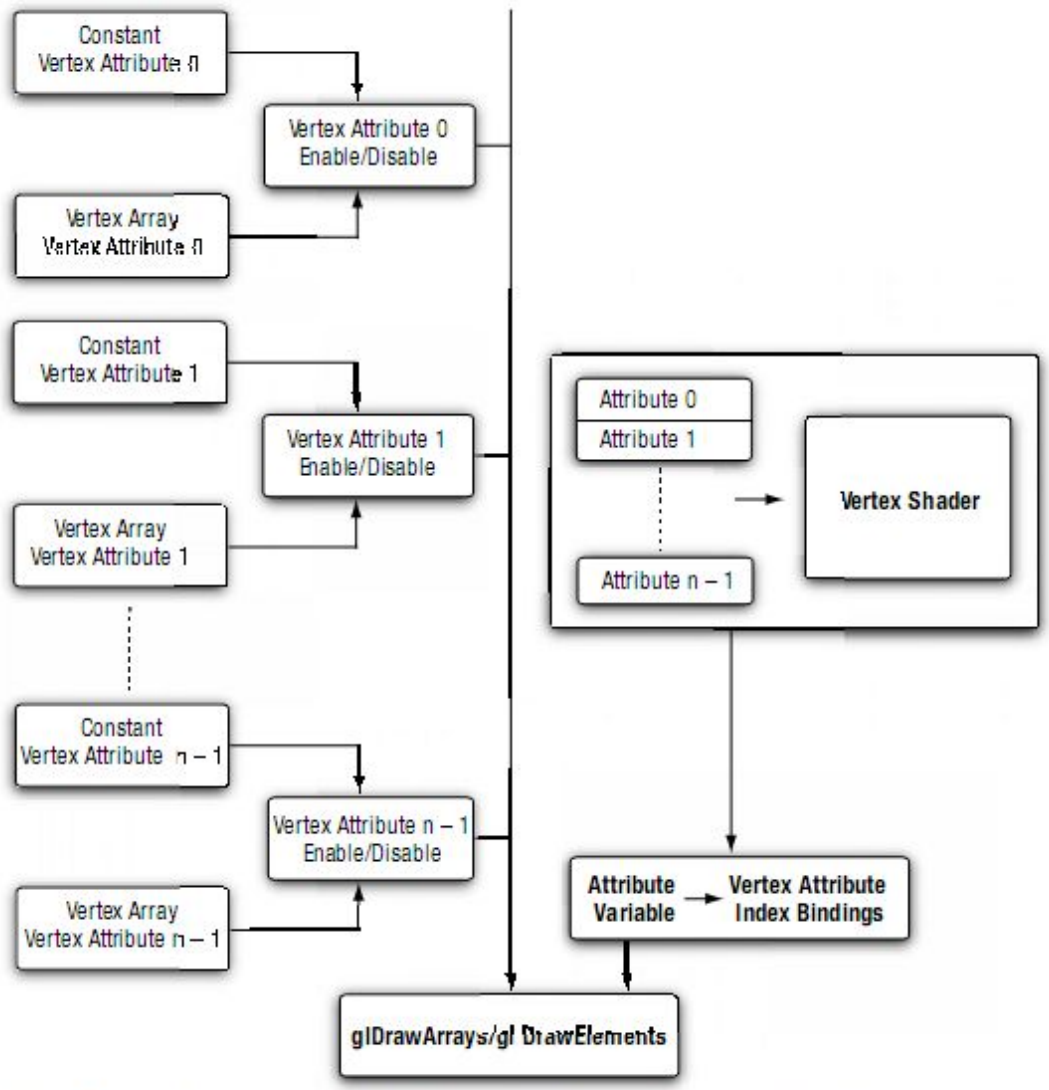


Figure 6-4 Specifying and Binding Vertex Attributes for Drawing a Primitive(s)

我们谈到顶点着色器里，顶点属性变量被使用 attribute 限定词指定，实际的激活的属性数能被使用 glGetProgamiv 查询，项目中激活的属性列表能被 glGetActiveAttrib 查询。我们也描述了激活的属性索引使用范围从 0 到 GL_ACTIVE_ATTRIBUTES –1，使能顶点属性和使用 glVertexAttrib*和 glVertexAttribPointer 指定一个常量或每个顶点值（像顶点数组）。现在我们描述怎么匹配这些属性索引到合适的在顶点着色器里声明的属性变量。这个匹配将允许正确的顶点数据被读到正确的顶点属性变量里。

图 6-4，描述一般被指定的顶点属性并绑定到着色器的属性名字。

有两种方法使能匹配顶点属性索引到属性变量名字。如下：

OpenGL ES 2.0 按顺序绑定顶点属性索引到属性变量名字。

应用程序绑定顶点属性索引到属性变量名字。

顶点着色器中使用 glBindAttribLocation 绑定一个普通的顶点属性索引到属性变量，这个绑定在项目下次链接时有效，在当前链接的项目里不会改变。

```
void    glBindAttribLocation(GLuint program, GLuint index,
                             const GLchar *name)
```

program　项目对象名字
index　　一般的顶点属性索引
name　　属性变量名字

如果一个名字前面被绑定过，它分配的绑定被一个索引取代。glBindAttribLocation 甚至能在顶点着色器被连接到一个项目对象前使用。这意味着这个调用能被使用去绑定任何属性名字。属性名字不存在或者没激活的绑定在项目里被忽略。

另一个选择是让 OpenGL ES 2.0 绑定属性变量名到一个普遍的顶点属性索引。这种绑定在项目链接时被执行。在链接阶段 OpenGL ES 2.0 为每个属性变量执行下面的步骤；

检查每个属性变量，是否已通过 glBindAttribLocation 绑定。如果已绑定，指定的合适的属性索引被使用。如果没有，编译工具将产生一个一般的顶点属性索引。

这个分配依赖编译工具，不同的编译工具将产生不同。应用能够使用 glGetAttribLocation 查询分配的绑定。

```
GLint    glGetAttribLocation(GLuint program,
                             const GLchar *name)
```

program　　项目对象
name　　属性变量名字

glGetAttribLocation 返回一个绑定属性变量名字的普通属性索引，当一个项目对象被项目链接时。如果那个名字不是激活的属性变量，或者项目不是一个有效项目对象，或者没有链接成功，返回-1，指示一个无效的属性索引。

顶点缓冲区对象

顶点矩阵使用的顶点数据被存储在客户内存中，当调用 glDrawArrays 或 glDrawElements 时，这些数据从客户内存被拷贝到绘图内存。这两个函数在第 7 章基本装配和光栅化描述，如果不用每次绘图时调用拷贝直接存储数据在绘图内存中将是更好的选择。这将显著提高渲染执行，减小内存带宽和电力消耗，所有这些对便携设备是非常重要的。这就需要用到顶点缓冲区对象来帮忙。顶点缓冲区对象允许 OpenGL ES 2.0 应用去分配和存储顶点数据到高效的绘图内存，从这里直接执行渲染。不仅仅是顶点数据，甚至是描述基元的顶点索引能使用 glDrawElements 去存储。

OpenGL ES 支持两种类型的缓冲区对象，array buffer objects 和 element array buffer objects。矩阵缓冲区对象指使用 GL_ARRAY_BUFFER 创建的存储顶点数据的缓冲区对象。元素矩阵缓冲区对象是使用 GL_ELEMENT_ARRAY_BUFFER 创建的存储元素索引的缓冲区对象。

注意：为更好地执行程序，我们推荐 OpenGL ES 2.0 应用尽可能的对顶点属性使用顶点缓冲区和元素索引。

我们使用缓冲区对象前，需要分配缓冲区对象重新装载顶点数据和元素索引到合适的缓冲区对象。这在例 6-4 的代码里展示：

例 6-4　创建和绑定顶点缓冲区对象

```
void    initVertexBufferObjects(vertex_t *vertexBuffer,
                                GLushort *indices,
                                GLuint numVertices, GLuint numIndices
                                GLuint *vboIds)
```

```
{
    glGenBuffers(2, vboIds);
    glBindBuffer(GL_ARRAY_BUFFER, vboIds[0]);
    glBufferData(GL_ARRAY_BUFFER, numVertices * sizeof(vertex_t),
                 vertexBuffer, GL_STATIC_DRAW);
    // bind buffer object for element indices
    glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIds[1]);
    glBufferData(GL_ELEMENT_ARRAY_BUFFER,
                 numIndices * sizeof(GLushort),indices,
                 GL_STATIC_DRAW);
}
```

代码创建了两个缓冲区对象：一个存储实际的顶点属性数据，一个存储基元组成的元素索引。这个例子中 glGenBuffers 函数被调用创建两个未使用的缓冲区对象名字存储到 vboIds 中，这两个未使用的缓冲区对象被使用创建一个矩阵缓冲区对象和一个元素矩阵缓冲区对象。矩阵缓冲区对象被用来存储一个或更多基元的顶点属性数据。元素矩阵缓冲区对象被用来存储基元索引。实际的矩阵或元素数据使用 glBufferData 指定。主要 GL_STATIC_DRAW 作为参数被 glBufferData 使用。应用怎么联系缓冲区将在后面章节描述。

void glGenBuffers(GLsizei n, GLuint *buffers)

n 返回的缓冲区名字数目

buffers 分配的缓冲区目标返回的 n 个入口点的矩阵的指针

glGenBuffers 分配 n 个缓冲区对象名字，在 buffers 返回他们。被 glGenBuffers 返回的缓冲区对象名是未使用的非 0 整型数。0 被 OpenGL ES 保留不分配。应用使用 0 去查询或修改缓冲区对象将产生一个对应的错误。

glBindBuffer 函数被使用使能一个缓冲区对象成为当前矩阵缓冲区对象或当前元素矩阵缓冲区对象。一个缓冲区对象被第一次使用 glBindBuffer 绑定时，缓冲区对象被分配合适的默认状态，如果分配成功，分配的对象被绑定成为渲染上下文的当前矩阵缓冲区对象或当前元素矩阵缓冲区对象。

void glBindBuffer(GLenum target, GLuint buffer)

target 被设定为 GL_ARRAY_BUFFER 或 GL_ELEMENT_ARRAY_BUFFER

buffer 缓冲区对象被指定为目标的当前对象

注意：glGenBuffers 在被使用 glBindBuffer 绑定前，不要求被指定缓冲区对象名，一个应用能指定一个未使用的缓冲区对象名字去绑定。但我们推荐使用 glGenBuffers 返回的名字而不是自己指定。

缓冲区对象能够被分类：

GL_BUFFER_SIZE 这指被 glBufferData 指定的缓冲区对象的数据大小，第一个被 glBindBuffer 绑定的对象是 0

GL_BUFFER_USAGE 指应用使用缓冲区对象存储数据的方法。在表 6-2 中描述，基本值是 GL_STATIC_DRAW。

表 6-2 缓冲区用法

Buffer Usage Enum	Description
GL_STATIC_DRAW	缓冲区对象数据一旦被应用程序指定，多次使用去绘制基元
GL_DYNAMIC_DRAW	缓冲区对象数据多次被应用程序指定，多次使用去绘制基元
GL_STREAM_DRAW	缓冲区对象数据一旦被应用程序指定，很少使用去绘制基元

像以前提到的那样，GL_BUFFER_USAGE 是 OpenGL ES 的提示不是保证。因此一个应用应该使用 usage 设定为 GL_STATIC_DRAW，分配缓冲区对象数据存储并且经常修改它。

被 OpenGL ES 缓冲区丢弃的 OpenGL 支持的 Usage 的枚举值

GL_STATIC_READ, GL_STATIC_COPY, GL_DYNAMIC_READ,GL_DYNAMIC_COPY, GL_STREAM_READ, 和 GL_STREAM_COPY 枚举值被 OpenGL 支持但不再被 OpenGL ES 支持，这是因为这些枚举值是从 GL 数据存储空间读回的内容指定的。OpenGL 允许应用读顶点缓冲区中的内容，但 OpenGL 舍弃了这些 API。没有读回缓冲区数据的机制，那些枚举值不再有效，不再被支持。

顶点矩阵数据和元素顶点矩阵数据被创建和初始化使用 glBufferData 函数

void glBufferData(GLenum target, GLsizeiptr size,
const void *data, GLenum usage)

target 被设定为 GL_ARRAY_BUFFER 或 GL_ELEMENT_ARRAY_BUFFER

size 缓冲区数据以字节（bytes）存储的大小

data 被程序支持的缓冲区数据指针

usage 应用怎样使用存储在缓冲区对象中数据的提示，细节看表 6-2

glBufferData 将基于 size 值存储数据。数据如果为空，代表存储的值未初始化。如果数据是有效的指针，data 的内容将被拷贝到存储数据的地方。缓冲区对象数据的内容使用 glBufferSubData 初始化或更新。

void glBufferSubData(GLenum target, GLintptr offset,
GLsizeiptr size, const void *data)

target 能被设定为 GL_ARRAY_BUFFER 或 GL_ELEMENT_ARRAY_BUFFER

offset, 缓冲区数据存储的偏置值

size 被修改的存储数据的字节数

data 需要被复制到缓冲区对象的客户区数据的指针

使用 glBufferData 或 glBufferSubData 初始化或更新缓冲区对象数据存储后，客户数据存储区不再需要，能够被释放。对静态几何图形，应用能够释放客户存储区减小系统内存消耗，对动态几何图形这是不可能的。

现在我们看使用和不使用缓冲区对象来画基元。例 6-5 描述了这两种情况，注意建立顶点属性的代码是很简单的，这个例子中我们对所有的顶点属性使用相同的缓冲区对象。当 GL_ARRAY_BUFFER 缓冲区对象被使用的时候，glVertexAttribPointer 的指针参数从实际数据的指针到使用 glBufferData 分配的以字节为单位的顶点缓冲存储区偏置量。相似的如果一个有效的 GL_ELEMENT_ARRAY_BUFFER 目标被使用，glDrawElements 的 indices 从实际元素索引指针变为使用 glBufferData 分配的元素索引缓冲存储区偏置值。

例 6-5 使用和不使用顶点缓冲区对象绘制图形

```
#define VERTEX_POS_SIZE       3     // x, y and z
#define VERTEX_NORMAL_SIZE    3     // x, y and z
#define VERTEX_TEXCOORD0_SIZE 2     // s and t
#define VERTEX_POS_INDX       0
#define VERTEX_NORMAL_INDX    1
#define VERTEX_TEXCOORD0_INDX 2
//
// vertices     C pointer to a buffer that contains vertex attribute
                data
// vtxStride    C stride of attribute data / vertex in byte
```

```
// numIndices C number of indices that make up primitive
//                      drawn as triangles
// indices       - pointer to element index buffer.
//
void     drawPrimitiveWithoutVBOs(GLfloat *vertices, GLint vtxStride,
                                  GLint numIndices, GLushort *indices)
{
     GLfloat     *vtxBuf = vertices;
     glBindBuffer(GL_ARRAY_BUFFER, 0);
     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, 0);
     glEnableVertexAttribArray(VERTEX_POS_INDX);
     glEnableVertexAttribArray(VERTEX_NORMAL_INDX);
     glEnableVertexAttribArray{VERTEX_TEXCOORD0_INDX);
     glVertexAttribPointer(VERTEX_POS_INDX, VERTEX_POS_SIZE,
                           GL_FLOAT, GL_FALSE, vtxStride, vtxBuf);
     vtxBuf += VERTEX_POS_SIZE;
     glVertexAttribPointer(VERTEX_NORMAL_INDX, VERTEX_NORMAL_SIZE,
                           GL_FLOAT, GL_FALSE, vtxStride, vtxBuf);
     vtxBuf += VERTEX_NORMAL_SIZE;
     glVertexAttribPointer(VERTEX_TEXCOORD0_INDX,
                           VERTEX_TEXCOORD0_SIZE, GL_FLOAT,
                           GL_FALSE, vtxStride, vtxBuf);
     glBindAttribLocation(program, VERTEX_POS_INDX, "v_position");
     glBindAttribLocation(program, VERTEX_NORMAL_INDX, "v_normal");
     glBindAttribLocation(program, VERTEX_TEXCOORD0_INDX,
                          "v_texcoord");
     glDrawElements(GL_TRIANGLES, numIndices, GL_UNSIGNED_SHORT,
                    indices);
}
void     drawPrimitiveWithVBOs(GLint numVertices, GLfloat *vtxBuf,
                               GLint vtxStride, GLint numIndices,
                               GLushort *indices)
{
     GLuint     offset = 0;
     GLuint     vboIds[2];
     // vboIds[0] C used to store vertex attribute dat
     // vboIds[1] C used to store element indice
     glGenBuffers(2, vboIds);
     glBindBuffer(GL_ARRAY_BUFFER, vboIds[0]);
     glBufferData(GL_ARRAY_BUFFER, vtxStride * numVertices,
                  vtxBuf, GL_STATIC_DRAW);
     glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIds[1]);
     glBufferData(GL_ELEMENT_ARRAY_BUFFER,
                  sizeof(GLushort) * numIndices,
```

```
                    indices, GL_STATIC_DRAW);
    glEnableVertexAttribArray(VERTEX_POS_INDX);
    glEnableVertexAttribArray(VERTEX_NORMAL_INDX);
    glEnableVertexAttribArray{VERTEX_TEXCOORD0_INDX);
    glVertexAttribPointer(VERTEX_POS_INDX, VERTEX_POS_SIZE,
                          GL_FLOAT, GL_FALSE, vtxStride,
                          (const void*)offset);
    offset += VERTEX_POS_SIZE * sizeof(GLfloat);
    glVertexAttribPointer(VERTEX_NORMAL_INDX, VERTEX_NORMAL_SIZE,
                          GL_FLOAT, GL_FALSE, vtxStride,
                          (const void*)offset);
    offset += VERTEX_NORMAL_SIZE * sizeof(GLfloat);
    glVertexAttribPointer(VERTEX_TEXCOORD0_INDX,
                          VERTEX_TEXCOORD0_SIZE,
                          GL_FLOAT, GL_FALSE, vtxStride,
                          (const void*)offset);
    glBindAttribLocation(program, VERTEX_POS_INDX, "v_position");
    glBindAttribLocation(program, VERTEX_NORMAL_INDX, "v_normal");
    glBindAttribLocation(program, VERTEX_TEXCOORD0_INDX,
                        "v_texcoord");
    glDrawElements(GL_TRIANGLES, numIndices, GL_UNSIGNED_SHORT, 0);
    glDeleteBuffers(2, vboIds);
}
```

例 6-5 我们使用一个缓冲区对象去存储所有的顶点数据，这代表着像例 6-1 描述的存储顶点属性方法中的结构矩阵的方法。这是可能的对每个顶点属性有一个缓冲区对象。在例 6-2 描述的矩阵结构存储顶点属性的方法，例 6-6 描述怎么使用 drawPrimitive 以 VBOs 参数对每个顶点属性查找独立的缓冲区对象。

例 6-6 每个属性绘制一个缓冲区对象

```
#define VERTEX_POS_SIZE       3     // x, y and z
#define VERTEX_NORMAL_SIZE    3     // x, y and z
#define VERTEX_TEXCOORD0_SIZE 2     // s and t
#define VERTEX_POS_INDX       0
#define VERTEX_NORMAL_INDX    1
#define VERTEX_TEXCOORD0_INDX 2
//
// numVertices C number of vertice
// vtxBuf C an array of pointers describing attribute dat
// vtxStrides C an array of stride values for each attribut
// numIndices C number of element indices of primitiv
// indices C actual element index buffe
//
void    drawPrimitiveWithVBOs(GLint numVertices,
                              GLfloat **vtxBuf, GLint *vtxStrides,
                              GLint numIndices, GLushort *indices)
```

```
{
    GLuint     vboIds[4];
    // vboIds[0] C used to store vertex positio
    // vboIds[1] C used to store vertex norma
    // vboIds[2] C used to store vertex texture coordinate
  // vboIds[3] C used to store element indice
  glGenBuffers(4, vboIds);
  glBindBuffer(GL_ARRAY_BUFFER, vboIds[0]);
  glBufferData(GL_ARRAY_BUFFER, vtxStrides[0] * numVertices,
                vtxBuf[0], GL_STATIC_DRAW);
  glBindBuffer(GL_ARRAY_BUFFER, vboIds[1]);
  glBufferData(GL_ARRAY_BUFFER, vtxStrides[1] * numVertices,
                vtxBuf[1], GL_STATIC_DRAW);
  glBindBuffer(GL_ARRAY_BUFFER, vboIds[2]);
  glBufferData(GL_ARRAY_BUFFER, vtxStrides[2] * numVertices,
                vtxBuf[2], GL_STATIC_DRAW);
  glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, vboIds[3]);
  glBufferData(GL_ELEMENT_ARRAY_BUFFER,
                sizeof(GLushort) * numIndices,
                indices, GL_STATIC_DRAW);
  glBindBuffer(GL_ARRAY_BUFFER, vboIds[0]);
  glEnableVertexAttribArray(VERTEX_POS_INDX);
  glBindBuffer(GL_ARRAY_BUFFER, vboIds[1]);
  glEnableVertexAttribArray(VERTEX_NORMAL_INDX);
  glBindBuffer(GL_ARRAY_BUFFER, vboIds[2]);
  glEnableVertexAttribArray{VERTEX_TEXCOORD0_INDX);
  glVertexAttribPointer(VERTEX_POS_INDX, VERTEX_POS_SIZE,
                        GL_FLOAT, GL_FALSE, vtxStrides[0], 0);
  glVertexAttribPointer(VERTEX_NORMAL_INDX, VERTEX_NORMAL_SIZE,
                        GL_FLOAT, GL_FALSE, vtxStrides[1], 0);
  glVertexAttribPointer(VERTEX_TEXCOORD0_INDX,
                        VERTEX_TEXCOORD0_SIZE,
                        GL_FLOAT, GL_FALSE, vtxStrides[2], 0);
    glBindAttribLocation(program, VERTEX_POS_INDX, "v_position");
    glBindAttribLocation(program, VERTEX_NORMAL_INDX, "v_normal");
    glBindAttribLocation(program, VERTEX_TEXCOORD0_INDX,
                         "v_texcoord");
    glDrawElements(GL_TRIANGLES, numIndices, GL_UNSIGNED_SHORT, 0);
    glDeleteBuffers(4, vboIds)
}
```

应用使用缓冲区对象处理完后，能够使用 glDeleteBuffers 函数删除。

void　glDeleteBuffers(GLsizei n, const GLuint *buffers)

n　　将要删除的缓冲区对象数目

buffers　将要删除的缓冲区对象的包含 n 个入口的矩阵

glDeleteBuffers 删除缓冲区中指定的缓冲区对象。一旦一个缓冲区对象被删除，它能够被其它的基元的顶点属性或元素索引的缓冲区对象所使用。

当你看这些例子，使用顶点缓冲区对象是非常容易的，要求很少的额外工作去做。这工作包括支持顶点缓冲区是非常值得的，考虑到这个执行需要的提供的。在下章我们讨论使用 glDrawArrays 和 glDrawElements 绘制基元，和 OpenGL ES 2.0 怎样基元装配及光栅化管道阶段。

匹配缓冲区对象

OES_map_buffer 扩展允许应用去匹配和不匹配一个顶点缓冲区对象的数据存储到应用程序地址空间。如果匹配函数返回这个地址存储的指针。指针能被应用程序使用更新缓冲区对象的内容。如果不匹配函数被使用指示更新已经被完成，去释放匹配的指针。下面的段落提供这些函数的深度描述和性能细节技巧。

glMapBufferOES 函数匹配一个顶点缓冲区对象的数据存储地址到应用程序地址空间。

void *glMapBufferOES(GLenum target, GLenum access)

target 必须设定为 GL_ARRAY_BUFFER

access 仅仅能设为 GL_WRITE_ONLY_OES

glMapBufferOES 返回这个存储的指针。匹配操作提供只写的接口到顶点缓冲区对象的地址存储。一个只读的内存地址匹配被返回的指针是未知的。依靠操作系统的性能，读操能产生一个错误或返回虚伪的地址。如果缓冲区匹配 glMapBufferOES 将返回一个有效的指针或返回 0.

glMapBufferOES 函数解除一个先前匹配的缓冲区

GLboolean glUnmapBufferOES(GLenum target)

target 必须设定为 GL_ARRAY_BUFFER

如果不匹配操作成功，glUnmapBufferOES 返回真。当成功不匹配已经被执行后，glMapBufferOES 返回的指针不能再被使用。当缓冲区被匹配后，如果顶点缓冲区对象的数据存储崩溃，glUnmapBufferOES 返回 GL_FALSE。这能在屏幕的变化反映出来，多屏被 OpenGL ES 上下文使用，或者一个内存溢出事件引起内存匹配被丢弃 [1]。

性能技巧

glMapBufferOES 仅仅被使用于整个缓冲区被更新的情况。不提倡使用 glMapBufferOES 去更新一个子区域，glMapBufferOES 没有机制去指定子区域。

即使使用 glMapBufferOES 更新整个缓冲区，这个操作对比 glBufferData 仍然是昂贵的。使用 glMapBufferOES，应用程序得到一个指向缓冲区对象数据的指针。无论如何，这个缓冲区可能被应用程序用于先前排队的渲染函数。如果缓冲区被当前使用，GPU 必须等待先前的渲染函数使用这个缓冲区完成，再返回指针。如果整个区域被更新，这就不需要等待。我们已经谈到 OpenGL ES 中没有区分 glMapBufferOES 函数更新整个区域或部分区域数据的机制。对应用程序来说指示整个区域被更新是一个方法。这能使用 glBufferData 设定参数为 NULL，跟着使用 glMapBufferOES 来完成。使用 NULL 调用 glBufferData 告诉 OpenGL ES 应用，先前的缓冲区是无效的，成功调用 glMapBufferOES 能被应用正确优化。

1 如果屏幕变成更大的宽和高的屏幕，运行中的像素的位数和匹配的内存被释放。注意这在嵌入式设备中不是经常遇到的。backing 存储在大多数的嵌入式设备中不被支持，所以内存溢出将导致内存被自由释放，被其他重要的应用重新分配。

7 基元装配和光栅化

本章我们描述基元类型，OpenGL ES 支持的几何图形和怎么绘制它们。当基元的顶点被着色器产生后，进入基元装配阶段。在基元装配阶段，剪裁、透视分割、视口转化被执行。这些操作被详细讨论。然后我们关注光栅化阶段，光栅化是转化基元成为一系列二维片段，这是片段着色器的工作。二维片段绘制的像素能够被绘制到屏幕。

第 8 章顶点着色器描述顶点着色器的细节，第 9 章贴图，第 10 章片段着色器描述应用片段着色器产生光栅化的过程。

基元

基元是使用 glDrawArrays 和 glDrawElements 能够绘制的图形。基元是一系列用顶点位置、颜色、贴图坐标等信息描述的顶点。

OpenGL ES 2.0 能绘制的基元有：

三角形

直线

点

三角形

三角形是 3D 应用中最普遍的渲染几何物体的方法。被 OpenGL ES 2.0 支持的三角形基元有 GL_TRIANGLES,GL_TRIANGLE_STRIP, 和 GL_TRIANGLE_FAN。图 7-1 显示了支持的三角形基元类型的例子

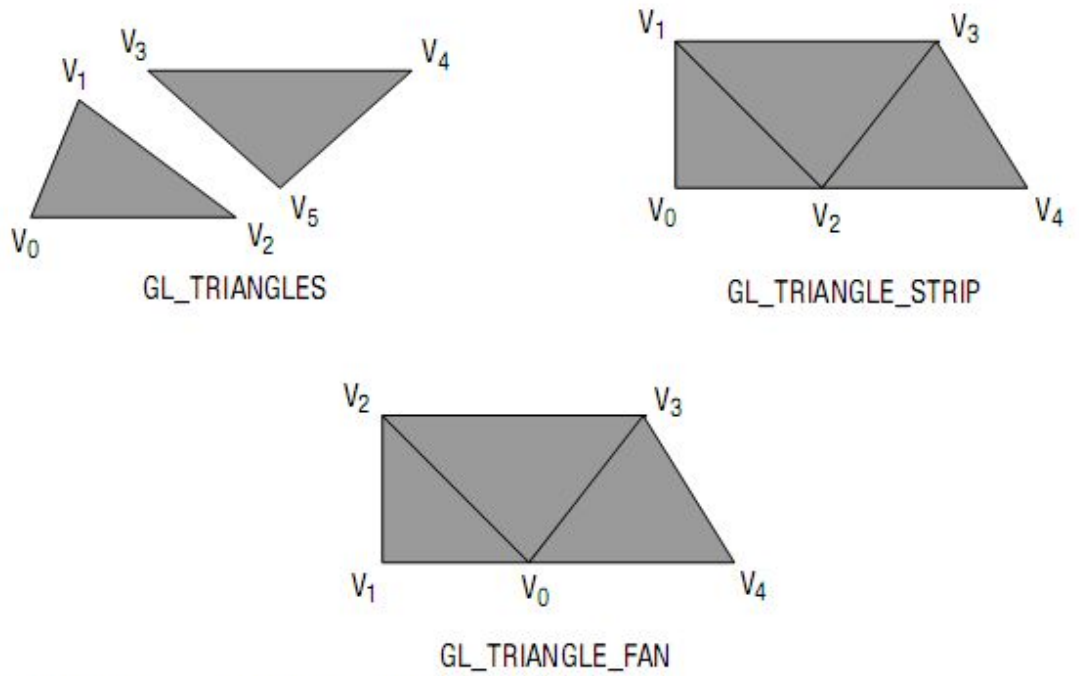


Figure 7-1 Triangle Primitive Types

GL_TRIANGLES 描述一系列独立的三角形，图 7-1 中，使用顶点(V_0,V_1 ,V_2) 和(V_3 ,V_4,V_5)，绘制了两个独立的三角形。一共有 n/3 的三角形会被绘制，n 是 glDrawArrays or glDrawElements 的参数 count 的索引数目。

GL_TRIANGLE_STRIP 绘制一系列相连的三角形。例子中三个三角形被绘制，(V_0,V_1 ,V_2)、(V_2,V_1 ,V_3)和(V_2,V_3 ,V_4)（注意顺序）。一共有 n-2 个三角形被绘制。

GL_TRIANGLE_FAN 也绘制了一系列相连的三角形，例子中三个三角形被绘制，(V_0,V_1 ,V_2)、(V_0,V_2 ,V_3)和(V_0,V_3 ,V_4)（注意顺序）。一共有 n-2 个三角形被绘制。

直线

被 OpenGL ES 支持的直线基元有 GL_LINES,GL_LINE_STRIP,和 GL_LINE_LOOP。图 7-2 显示了直线基元的类型

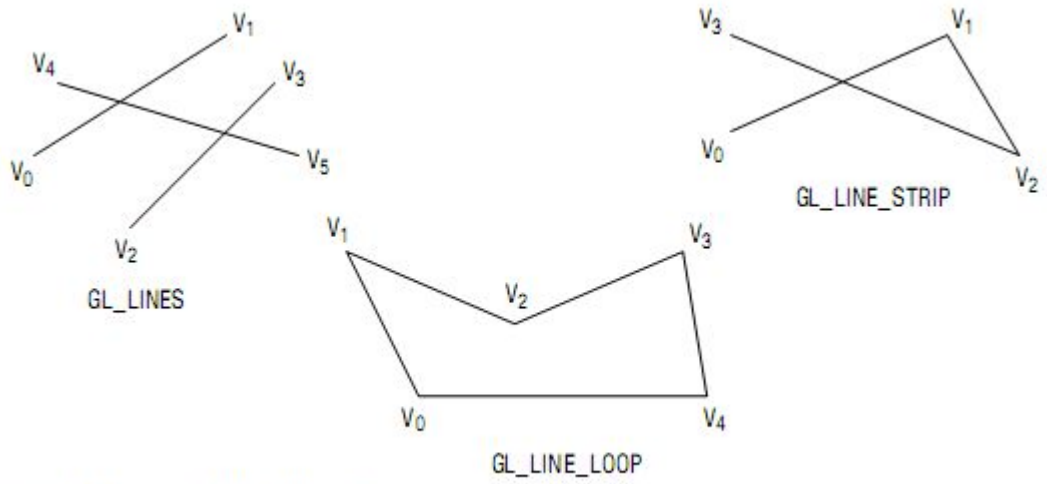


Figure 7-2 Line Primitive Types

GL_LINES 绘制一系列不相连的直线，例子中三个不相连的直线被绘制，(V_0,V_1),(V_2,V_3) 和 (V_4,V_5).总数 n/2 的线段被绘制，n 是 glDrawArrays 或 glDrawElements 的参数 count 的索引数目。

GL_LINE_STRIP 绘制一系列相连的直线，例子中三个不相连的直线被绘制，(V_0,V_1),(V_1,V_2)和 (V_2,V_3)。总数 n-1 的线段被绘制，n 是 glDrawArrays or glDrawElements 的参数 count 的索引数目。

GL_LINE_LOOP 和 GL_LINE_STRIP 的情况类似，除了最后的线段绘制 V_{n-1} 到 V_0。例子中(V_0 ,V_1), (V_1,V_2), (V_2,V_3), (V_3 ,V_4), and (V_4 ,V_0)被绘制。总数 n-1 的线段被绘制，n 是 glDrawArrays or glDrawElements 的参数 count 的索引数目。

直线宽使用 glLineWidth 指定。

```
void     glLineWidth(GLfloat width)
width    specifies the width of the line in pixels. The default width is 1.0
```

glLineWidth 设定的线宽被 OpenGL ES 2.0 强制执行，线宽范围可以使用下面的函数查询，没有对线宽大于 1 的直线支持的要求。

```
GLfloat     lineWidthRange[2];
glGetFloatv(GL_ALIASED_LINE_WIDTH_RANGE, lineWidthRange);
```

点纹理

OpenGL ES 支持的点纹理是 GL_POINTS。每个具体的点被绘制。点是特殊的，当绘制他们为点而不是正方形，使用的渲染效率被影响。点是被指定位置和半径的对齐屏幕的矩形。位置是方形的中心，半径被使用计算四个正方形的坐标。

gl_PointSize 是在着色器里用来做点半径的变量。顶点着色器联系的点基元的输出 gl_PointSize 是个重要的量，另外点的尺寸值如果未定义，将导致绘图错误。OpenGL ES 2.0 支持着色器的输出 gl_PointSize 值被强制为点尺寸范围的别名。范围能够被使用以下函数查询。

```
GLfloat     pointSizeRange[2];
glGetFloatv(GL_ALIASED_POINT_SIZE_RANGE, pointSizeRange);
```

默认，OpenGL ES 2.0 把窗口坐标(0, 0)定义到（左，底）。但对点纹理，坐标原点是（左，

顶）。

gl_PointCoord 是仅仅在片段着色器内的变量，当基元被写入为点纹理时。gl_PointCoord 被声明为 vec2 变量，使用 mediump 精度值。gl_PointCoord 值从 0.0 到 1.0，代表着从左到右和从顶到底，图 7-3 解释这过程。

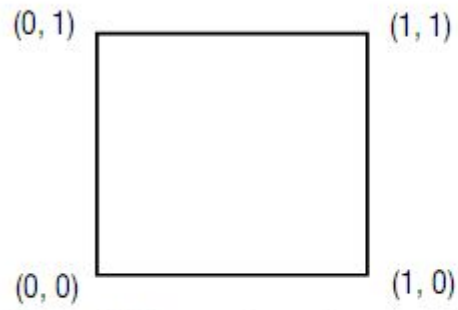


Figure 7-3 gl_PointCoord Values

下面的代码解释 gl_PointCoord 使用贴图坐标绘制点纹理贴图。

```
uniform sampler2D s_texSprite;
void
main(void)
{
    gl_FragColor = texture2D(s_texSprite, gl_PointCoord);
}
```

绘制基元

OpenGL ES 里有两个函数用来绘制基元

void glDrawArrays(GLenum mode, GLint first, GLsizei count)

mode 指定绘制的类型，有效值为：

GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN

first 指定使能的顶点矩阵中顶点索引的开始

count 指定被绘制的索引数目

void glDrawElements(GLenum mode, GLsizei count,
GLenum type, const GLvoid *indices)

Mode 指定绘制的类型，有效值为：

GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_TRIANGLES
GL_TRIANGLE_STRIP

GL_TRIANGLE_FAN

count　指定绘制的索引数目

type　指定存储在索引表中的索引元素类型，有效值为：

GL_UNSIGNED_BYTE

GL_UNSIGNED_SHORT

GL_UNSIGNED_INT 可选 (仅仅 OES_element_index_uint 扩展可用时被使用)

indices　指定元素索引存储位置的指针

glDrawArrays 使用元素索引表中的 first 到 first + count-1 作顶点，使用 mode 作类型绘制基元。glDrawArrays(GL_TRIANGLES, 0, 6)将绘制两个三角形，一个是三角形的索引是(0,1,2)，另一个索引是(3,4,5)。glDrawArrays(GL_TRIANGLE_STRIP, 0, 5)将绘制三个三角形，第一个是(0,1,2)，第二个是(2,1,3)，第三个是(2,3,4)。

如果你有一列元素列表和基元描述，而且几何图形的顶点不被分享，使用 glDrawArrays 将是好的选择。但在游戏中和其它 3D 应用中常常用到元素索引不是顺序的，顶点被其它三角形分享的多三角形网格。

考虑 7-4 的立方体，使用 glDrawArrays 绘制代码如下：

```
#define VERTEX_POS_INDX 0
#define NUM_FACES           6
GLfloat vertices[] = {    };    // (x, y, z) per verte
glEnableVertexAttribArray(VERTEX_POS_INDX);
glVertexAttribPointer(VERTEX_POS_INDX, 3, GL_FLOAT, GL_FALSE,
                              0, vertices);
for (i=0; i<NUM_FACES; i++)
{
    glDrawArrays(GL_TRIANGLE_FAN, first, 4);
    first += 4;
}
```

或

```
glDrawArrays(GL_TRIANGLES, 0, 36);
```

绘制这个立方体，每一面都要调用 glDrawArrays。被分享的顶点需要重复调用。不是使用了 8 个顶点，而是指定了 24 个（使用 GL_TRIANGLE_FAN)或 36 个（使用 GL_TRIANGLES)顶点，这是无效的方法。

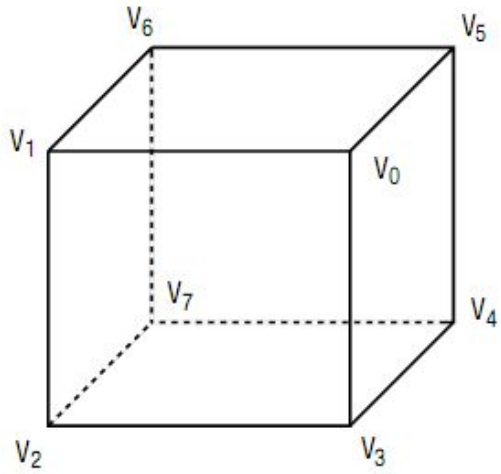


Figure 7-4　Cube

我们现在使用 glDrawElements 绘制同样的立方体。

```
#define VERTEX_POS_INDX 0
GLfloat vertices[] = {    };// (x, y, z) per verte
GLubyte indices[36] = { 0, 1, 2, 0, 2, 3,
                        0, 3, 4, 0, 4, 5,
                        0, 5, 6, 0, 6, 1,
                        7, 6, 1, 7, 1, 2,
                        7, 4, 5, 7, 5, 6,
                        7, 2, 3, 7, 3, 4 };
glEnableVertexAttribArray(VERTEX_POS_INDX);
glVertexAttribPointer(VERTEX_POS_INDX, 3, GL_FLOAT, GL_FALSE,
                      0, vertices);
glDrawElements(GL_TRIANGLES, sizeof(indices)/sizeof(GLubyte),
               GL_UNSIGNED_BYTE, indices);
```

虽然可以使用 glDrawElements 和 glDrawArrays 都可以绘制三角形，但在 GPU 上，因为很多原因 glDrawElements 比 glDrawArrays 更快。例如当顶点重复时使用 glDrawElements 顶点属性数据的量更少。这也让你需要更低的内存需求和带宽需要。

性能技巧

应用应该确认 glDrawElements 将调用尽可能多的基元。绘制 GL_TRIANGLES 将是容易的。但如果有三角形网眼或者扇形，为每个三角形网眼使用 glDrawElements 独立绘制，通过对退化的三角形增加元素索引，将这些三角形联系到一起。退化三角形是有有两个或更多顶点共用的三角形。GPU 很容易觉察和拒绝退化三角形，于是就可以让我们使用 GPU 渲染一个排列好的大基元。

需要对网眼（或退化三角形）增加的元素索引的数目依据于三角形的网眼是否是三角形的扇形或条带型以及每个带定义的索引数目。我们需要保证从一个三角形连接到另一个三角形的过程中那些明显的起到联系作用的网眼的顺序是合适的。（胡乱翻译的，看不太懂）

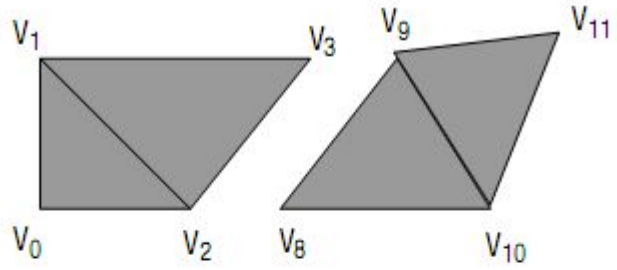
连接独立的三角形时，我们需要检查最后一个三角形和下一个的三角形的连接带的顺序，像图 7-2，偶数个顶点三角形连接带和奇数个顶点的三角形连接带的顶点顺序是不同的。

两种情况需要被处理：

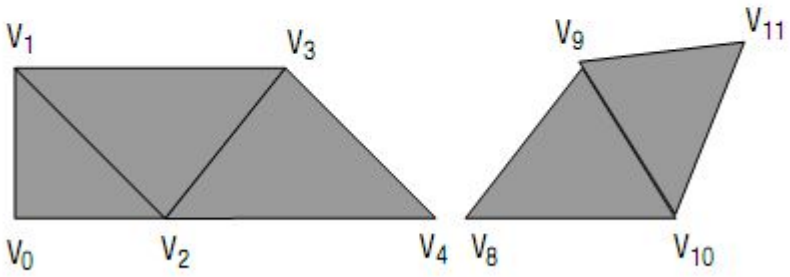
第一个三角形带的偶数三角形被连接到第二个三角形带中的第一个（因此叫偶数）三角形。

第一个三角形带的奇数三角形被连接到第二个三角形带中的第一个（因此叫偶数）三角形（完全不懂）。

图 7-5 显示绘制两种独立的三角形串的情况，我们使用 glDrawElements 来绘制这些带。



Opposite Vertex Order



Same Vertex Order

Figure 7-5 Connecting Triangle Strips

图 7-5 中相反顶点顺序的两个带，后面的三角形带和前面的三角形带被联系到一起，三角形带的索引值是各自独立的，分别是(0,1,2,3) 和 (8,9,10,11)。如果我们将两个元素列表联合起来，使用 glDrawElements 调用一次绘制两个带将是(0,1,2,3,**3,8**,8,9,10,11)。新的元素列表绘制了下面这些三角形(0,1,2), (2,1,3),**(2,3,3),(3,3,8),(3,8,8), (8,8,9)**,(8,9,10),(10,9,11)。黑体类型的三角形是退化三角形。黑体类型的元素列表代表两个索引列表结合后增加的索引。

图 7-5 中相同顶点顺序的两个带，后面的三角形带和前面的三角形带被联系到一起，三角形带的索引值是各自独立的，分别是(0,1,2,3,4) 和 (8,9,10,11)。如果我们将两个元素列表联合起来，使用 glDrawElements 调用一次绘制两个带将是(0,1,2,3,4,**4,**8,9,10,11)。新的元素索引列表绘制下面这些三角形(0,1,2),(2,1,3),(2,3,4),**(4,3,4), (4,4,8), (8,4,9)**, (8,9,10), (10,9,11)。黑体类型的三角形是退化三角形。黑体类型的元素列表代表两个索引列表结合后增加的索引。

注意增加的元素列表数目和退化三角形产生的变量数目依靠第一个带的顶点数目。用来保证于下一个带被联系时旋转的顺序。

研究后转换（post-transform）顶点缓冲区的大小决定怎样安排基元元素的索引可能是值得的。大多数的 GPU 提供一个 post-transform 顶点高速缓存。在一个顶点（提供它的元素目录或索引）被着色器执行前，检查顶点是否已经存在 post-transform 顶点高速缓存中。如果顶点已经存在 post-transform 顶点高速缓存中，顶点不被着色器执行。如果不在 post-transform 顶点高速缓存中，顶点将需要被着色器执行。使用 post-transform 缓存尺寸去决定元素目录怎么创建将有助于整体性能，将减少着色器去执行顶点被重新装载的时间。

基元装配

图 7-6 显示基元装配阶段。glDrawArrays 或 glDrawElements 提供的顶点被着色器执行。包括描述顶点位置的(x,y,z,w)值被顶点着色器转换。基元类型和顶点目录决定何种基元将被渲染。例 7-6 显示每个独立的基元（三角形、线和点）和它们对应的顶点，在基元装配阶段的操作。

我们讨论基元被光栅化前，需要懂得 OpenGL ES 2.0 使用的各种坐标系统。这需要弄懂顶点坐标在 OpenGL ES 2.0 管道的各个阶段发生的事情。

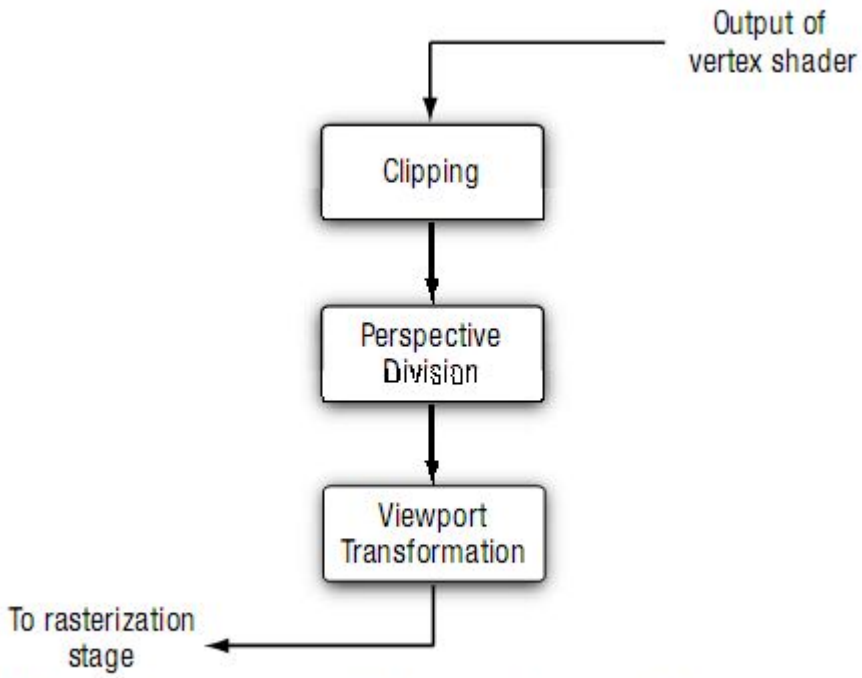


Figure 7-6 OpenGL ES Primitive Assembly Stage

坐标系统

图 7-7 显示了顶点通过顶点着色器和基元装配阶段的坐标系。顶点被输入 OpenGL ES 的对象或本地坐标系统空间。坐标空间是一个物体看起来的模型或存储。顶点着色器被执行后，顶点位置被认为是剪切坐标空间的值。顶点位置从本地坐标空间(物体自己的坐标空间)到剪切坐标空间的转化是通过在顶点着色器中装载合适的 uniform 矩阵来完成的。第 8 章描述怎样转化顶点位置从物体空间到剪切空间和在顶点着色器里装载合适的矩阵执行这个转化。

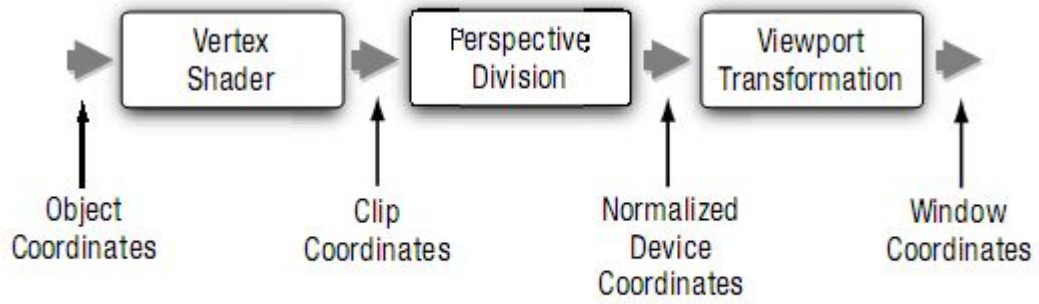


Figure 7-7 Coordinate Systems

剪切

顶点位置在顶点着色器执行后到达剪切坐标空间。剪切空间是齐次坐标空间(x_c, y_c , z_c , w_c)。顶点坐标在剪切空间(x_c, y_c , z_c , w_c)的定义后，被可见体（剪切体积）剪切。

在图 7-8 显示的剪切体积被使用 6 个剪切平面定义，分别是近和远的、左和右的、上和下的剪切平面。在剪切坐标系，剪切体积是：

$-w_c <= x_c <= w_c$

$-w_c <= y_c <= w_c$

$-w_c <= z_c <= w_c$

6 个检查帮助决定平面那些基元需要被剪切。

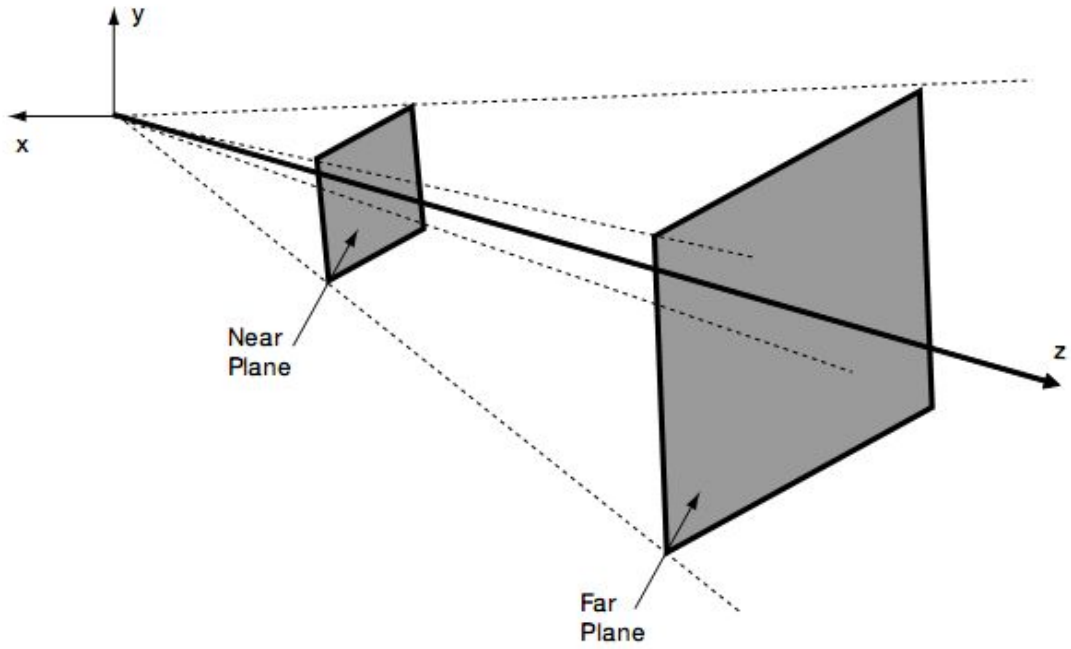


Figure 7-8 Viewing Volume

剪切阶段将剪切的剪切体积内的每个基元显示在图 7-8 内。这的每个基元是指使用 GL_TRIANGLES 绘制的独立的三角形，或者三角形带，或者扇形三角形，或者线段，或者封闭线，或者指定的点。

对每个基元类型，下面的操作被执行

剪切三角形—如果三角形完全在可视体积内，不用剪切。如果完全在体积外，被丢弃。如果部分在可视体积内，三角形被适当剪切，剪切操作产生新的顶点，剪切平面被规划为三角形扇。

剪切直线—如果直线完全在可视体积内，不用剪切。如果完全在体积外，被丢弃。如果部分在可视体积内，直线被适当剪切，剪切操作产生新的顶点。

剪切点—如果点位置在剪切面外或者远离剪切面，或者点的内切四边形在剪切体积外，点被丢弃。别的情况下它不改变，如果点的位置从内到外移动，点被剪去；或者相反。

基元被六个剪切面剪切后，顶点坐标再经过透视除法变成正常的设备坐标。正常的设备坐标范围在-1.0 到 +1.0。

注意：剪切操作对硬件来说是非常慢的（特别是线和三角形）。像图 7-8 中，一个基元必须被 6 个面剪切。基元部分在近和远平面中的将进行剪切操作。然而基元部分在 x 和 y 平面的不需要使用剪切。被渲染为视口后，它将大于指定的 glViewport 尺寸，在 x 和 y 平面的剔除变成剪切运算。这对 GPU 来说效率是很高的。大过 viewport 的区域叫 guard-band 区。虽然 OpenGL ES 不允许指定 viewport 区，但大多数（不是全部）OpenGL ES 支持 guard-band。

透视除法

透视除法使用剪切后的点(x_c, y_c , z_c , w_c)，投射它们到屏幕或可视区。这个投射执行(x_c, y_c , z_c)坐标除 w_c 。执行后我们得到正常的设备坐标（x_d，y_d，z_d）。他们称正常设备坐标是因为它们的值在[-1.0，1]范围内。（x_d，y_d）依靠可视区大小被转换到实际平面。（z_d）被转

换为屏幕 z 值，依靠使用 glDepthRange 指定的近和远的深度值。这些转化在视口转化阶段执行。

视口转换

视口转换使用下面的 API:

void glViewport(GLint x, GLint y, GLsizei w, GLsizei h)

x, y 指定视口下面边的屏幕坐标，单位是像素

w, h 指定视口宽和高，单位是像素，值必须> 0

从正常设备坐标（x_d，y_d，z_d）转换到窗口坐标（x_w，y_w，z_w）使用下面的转化：

$$\begin{bmatrix} x_w \\ y_w \\ z_w \end{bmatrix} = \begin{bmatrix} (w/2)x_d + o_x \\ (h/2)y_d + o_y \\ ((f-n)/2)z_d + (n+f)/2 \end{bmatrix}$$

其中 $o_x = (x + w)/2$, $o_y = (y + h)/2$, n 和 f 代表要求的深度范围。

n 和 f 深度范围使用下面的 API 指定：

void glDepthRange(GLclampf n, GLclampf f)

n, f 指定希望的深度范围，默认值 n 和 f 代表 0.0 和 1.0，值应该在(0.0, 1.0)范围内

glDepthRange 和 glViewport 指定的值被使用去转换顶点位置从正常设备坐标到窗口坐标。用作渲染的默认的视口被设定为 w = 宽 和 h = 高的窗口被 OpenGL ES 的函数创建，视窗使用 EGLNativeWindowType win 为参数调用 eglCreateWindowSurface 得到。

光栅化

图 7-9 显示了光栅化管道，当顶点被转换、基元被剪切后，光栅化管道使用基元像三角形、直线、或点产生基元相对应的片段。每个片段通过它在屏幕上的整数坐标(x, y)区别。片段代表着屏幕空间上一个坐标为(x, y)的像素，额外的片段数据被片段着色器处理产生片段颜色。这在第 9 章和第 10 章描述细节。

本章我们讨论应用能被使用去控制三角形、条和扇光栅化的选择项。

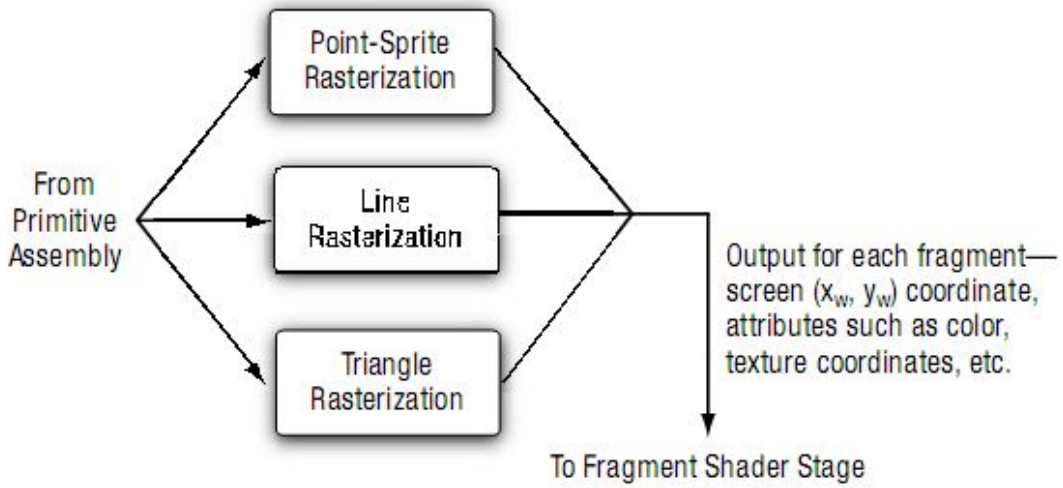


Figure 7-9 OpenGL ES Rasterization Stage

挑选

在三角形被光栅化前我们需要确定它们是否是前面（面向观察者）还是后面（背向观察

者）。挑选操作舍弃背向观察者的三角形。决定三角形是面向观察者还是背向观察者首先要知道三角形的方向。

三角形的方向定义为第一个顶点开始弯曲到第二个、第三个一直到最后一个顶点的路径顺序。图 7-10 描述了两个三角形顺时针和逆时针弯曲的顺序。

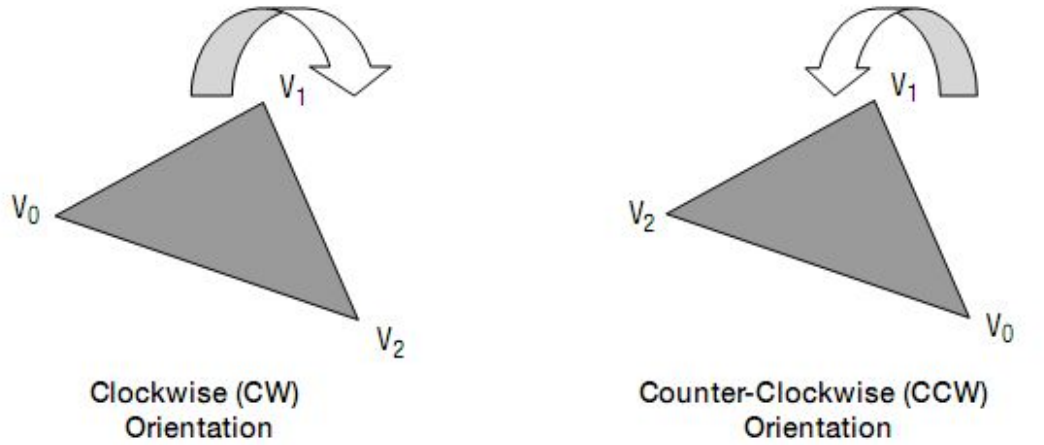


Figure 7-10 Clockwise and Counterclockwise Triangles

三角形的方向是通过计算窗口坐标系下三角形所占的面积得出的。我们需要去翻译计算出的三角形面积为顺时针（CW）或者逆时针（CCW）方向。计算出的三角形面积去匹配顺时针或者逆时针是通过使用下面的 API 指定：

void glFrontFace(GLenum dir)

dir 指定三角形正面方向，值为 GL_CW 或 GL_CCW. 默认值是 GL_CCW

我们讨论怎样计算三角形的方向。为决定三角形是否会被剔除，我们需要知道三角形的那个面将被剔除。这通过使用下面的 API 指定：

void glCullFace(GLenum mode)

mode 指定三角形的那个面将会被剔除，取值范围有 GL_FRONT, GL_BACK, 和 GL_FRONT_AND_BACK. 默认值是 GL_BACK

最后尤其重要的是，我们需要知道剔除操作是否需要执行。如果 GL_CULL_FACE 状态使能，则使用剔除，GL_CULL_FACE 通过使用下面的 API 使能或不使能。

void glEnable(GLenum cap)

void glDisable(GLenum cap)

初始化时,cap 设定为 GL_CULL_FACE，剔除不使能。

扼要重述，为剔除适当的三角形，OpenGL ES 必须使用 glEnable(GL_CULL_FACE)使能剔除。设定三角形的观察面方向使用 glFrontFace。

注意：剔除应该被使能去避免 GPU 去光栅化看不见的三角形而浪费时间。使能剔除将帮助提高整个 OpenGL ES 应用的执行效率。

多边形偏移

考虑我们绘制两个相互重叠的多边形的情况。你将看到图 7-11，那样的难看痕迹。那些痕迹称 Z-fighting artifacts，是因为三角形光栅化时精度限制造成的，这影响深度精度值产生碎片，导致了图 7-11 的痕迹。提高三角形光栅化参数的精度将深度值碎片变得更好，但并不能完全消除。

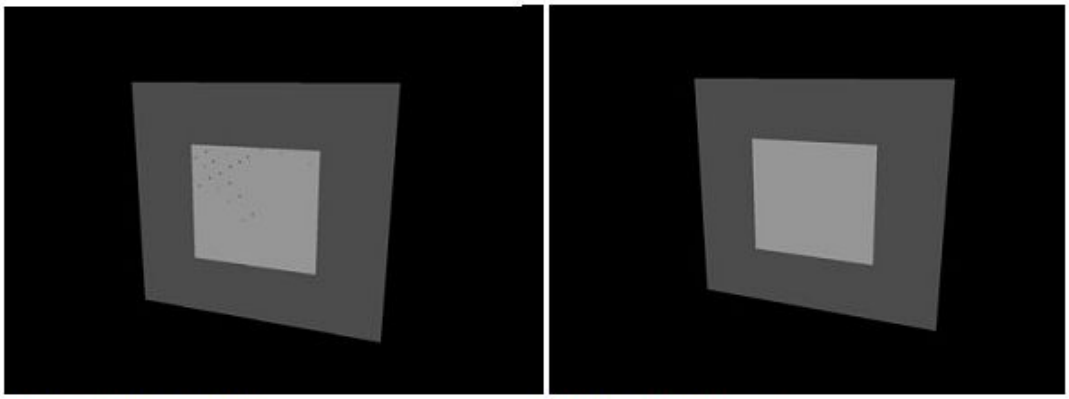


Figure 7-11 Polygon Offset

图 7-11 显示两个共面多边形的绘制。绘制没有使用多边形偏移的两个共面多边形的代码如下：

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// load vertex shader
// set the appropriate transformation matrices
// set the vertex attribute state
// draw the RED triangle
glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
// set the depth func to <= as polygons are coplanar
glDepthFunc(GL_LEQUAL);
// set the vertex attribute state
// draw the GREEN triangle
glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
```

为了避免图 7-11 中的痕迹，我们在深度测试前和深度值被写到深度缓冲区前需要增加一个 delta 去计算深度值，如果深度测试通过，深度的原始值不是原始的，深度值+ delta，将被存储到深度缓冲区。

多边形偏移使用下面的 API 设定：

void glPolygonOffset(GLfloat factor, GLfloat units)

深度偏移被计算：

深度值= m * factor + r * units

m 是最大的三角形值斜率被计算：

$$m = \sqrt{(\partial z/\partial x^2 + \partial z/\partial y^2)}$$

m 是计算出的$\{|\partial z/\partial x|, |\partial z/\partial y|\}$中的大的值。

OpenGL ES 在三角形光栅化阶段使用 $\partial z/\partial x$ 和$\partial z/\partial y$ 计算斜率。

r 是应用程序定义的常量，代表一个很小的值引起深度值的不同。

多边形偏移能使用 glEnable(GL_POLYGON_OFFSET_FILL) 使能或 glDisable(GL_POLYGON_OFFSET_FILL)不使能。

当多边形偏移使能，图 7-11 的渲染代码如下；

```
const float polygonOffsetFactor = -1.0f;
const float polygonOffsetUnits    = -2.0f;
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
// load vertex shader
// set the appropriate transformation matrices
// set the vertex attribute state
// draw the RED triangle
glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
// set the depth func to <= as polygons are coplanar
glDepthFunc(GL_LEQUAL);
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(polygonOffsetFactor, polygonOffsetUnits);
// set the vertex attribute state
// draw the GREEN triangle
glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
```

8 顶点着色器

这章描述 OpenGL ES 2.0 可编程顶点管道。图 8-1 阐述了 OpenGL ES 2.0 可编程管道。图 8-1 中的着色器部分指示了 OpenGL ES 2.0 可编程阶段。本章我们讨论顶点着色器阶段。顶点着色器能被应用于传统的基于顶点的操作，像通过矩阵转化位置、计算每个顶点的光照方程产生顶点颜色、产生贴图转换。

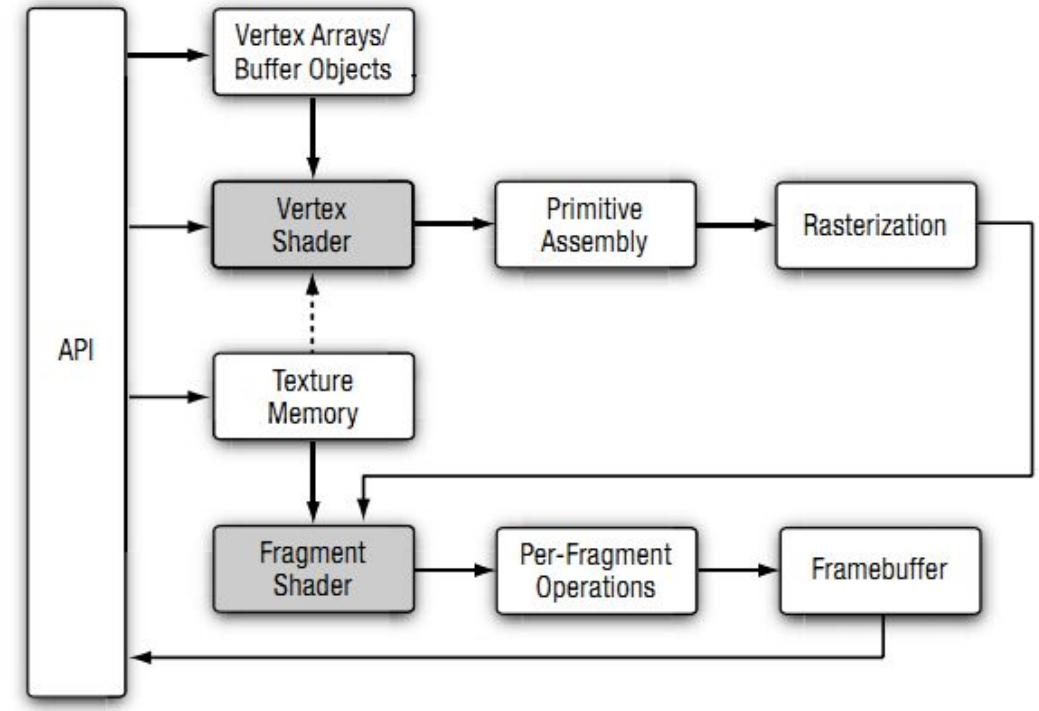


Figure 8-1 OpenGL ES 2.0 Programmable Pipeline

前面的章节，第 5 章 OpenGL ES 着色器语言，第 6 章顶点属性、顶点矩阵和缓冲对象讨论怎么去指定顶点属性和输入 uniform，也描述了 OpenGL ES 2.0 的着色器语言。第 7 章，基元装配和光栅化，讨论了顶点着色器的输出，也称变量，被光栅化阶段使用产生各个片段值，它们被输入片段着色器。本章我们整体来说顶点着色器的输入和输出。我们然后讨论 OpenGL ES 2.0 着色器语言的强制的限制，描述着色器时为了便于在各种编译工具中移植需要牢记的问题。然后通过几个例子描述怎么写顶点着色器。例子描述了一般的用法例如：使用可视模型转化顶点位置，产生顶点散射和镜面反射颜色的顶点光照，顶点贴图坐标产生，和顶点皮肤。我们希望这些例子能够使读者懂得怎么写顶点着色器，牢记让顶点着色器便于移植的规则，顶点着色器的最大的上限。更重要的是我们也描述 OpenGL ES 1.1 这种固定行为的顶点管道的着色器编译。这两种着色器将让读者懂得复杂的顶点着色器应该支持最早的使用 OpenGL ES 2.0 的硬件设备。

顶点着色器概述

顶点着色器提供一个普遍的顶点操作的编程方法。图 8-2 显示了顶点着色器的输入和输出。顶点着色器的输入包括：

属性—使用顶点矩阵形式的顶点数据

uniform—顶点着色器使用的常量数据

着色器项目—顶点着色器项目源码或者可执行部分，描述顶点将执行的操作

顶点着色器的输出被称为变量。在基本的光栅化阶段，对每个片段变量被计算出来输入

片段着色器。

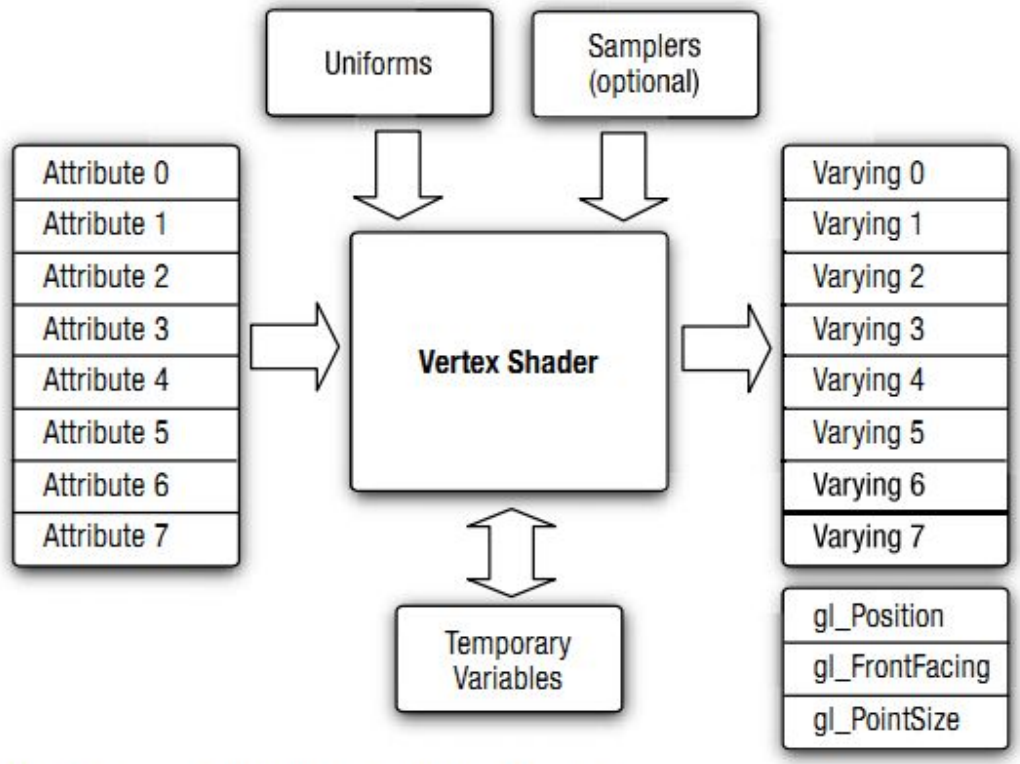


Figure 8-2 OpenGL ES 2.0 Vertex Shader

顶点着色器内置变量

顶点着色器内置变量能被着色器分类为顶点着色器特殊的输出变量，常量状态像深度范围，指定属性的数目最大值的常量，变量数目和 uniform 数目。

内建的特殊变量

顶点着色器内建的变量是顶点着色器输出后转变为片段着色器输入，或者被片段着色器输出。例如顶点着色器内建的特殊变量如下：

gl_Position—被使用在剪切坐标的顶点位置输出。gl_Position 被剪切和视口阶段使用，执行基元剪切和顶点位置从剪切坐标到屏幕坐标转换。如果顶点着色器不写到 gl_Position 中，gl_Position 值未定。gl_Position 是使用高质量精度的浮点型变量。

gl_PointSize—使用来写像素点的尺寸。当渲染点的时候 gl_PointSize 被使用。OpenGL ES 2.0 支持使用顶点着色器输出地 gl_PointSize 值去钳位虚点尺寸范围。gl_PointSize 是使用中质量精度的浮点型变量。

gl_FrontFacing—特殊的变量，虽然不直接被顶点着色器写，但被顶点着色器基于位置产生，渲染的基本类型，gl_FrontFacing 是布尔变量。

内建 uniform 状态

顶点着色器内仅仅的内建 uniform 状态变量是窗口坐标的深度范围。通过内建 uniform 的名字 gl_DepthRange,指定，下面声明一个类型是 gl_DepthRangeParameters 的 uniform。

```
struct gl_DepthRangeParameters {
    highp float near; // near Z
    highp float far;    // far Z
    highp float diff; // far - near
}
```

uniform gl_DepthRangeParameters gl_DepthRange;

内建常量

下面是顶点着色器中可用的的内建常量

```
const mediump int gl_MaxVertexAttribs                = 8;
const mediump int gl_MaxVertexUniformVectors         = 128;
const mediump int gl_MaxVaryingVectors               = 8;
const mediump int gl_MaxVertexTextureImageUnits      = 0;
const mediump int gl_MaxCombinedTextureImageUnits = 8;
```

内建常量描述下面的参数的最大值

gl_MaxVertexAttribs—顶点着色器属性数目最大值，最小值是 8.

gl_MaxVertexUniformVectors—在顶点着色器内 uniformvec4 输入的最大数目。ES 2.0 的最小数目是 128 个 vec4 输入。vec4 输入的数目在不同编译工具和着色器上是不同的。例如有的编译器计算使用者指定的文字值作为 uniform 限制。另一方面基于顶点着色器是否使用内建的超越函数，决定 uniform 或常量是否需要被编译器包含进去。当前应用没有找到 uniform 输入口数目的机制，但它能被特殊的顶点着色器使用。当编译器编译失败时，在编译器的日志里提供使用的 uniform 输入口数目的信息。无论如何，编译器日志返回比较详细的信息。我们提供一些指导详细，帮助使用顶点着色器里最大可用的 uniform 输入口的信息。

gl_MaxVaryingVectors—变量向量的最大数目，顶点着色器的能够用作输出的 vec4 入口的数目。最小数目是 8.

gl_MaxVertexTextureImageUnits—顶点着色器里可用的最大贴图单元的数目，最小值是 0，指示顶点着色器不支持贴图。

gl_MaxCombinedTextureImageUnits—这是顶点着色器和片段着色器支持的最大贴图单元的数目，最小值是 8.

对每个内建常量指定的值是 OpenGL ES 2.0 所有编译器必须支持的最小值。支持超过这个值是允许的。实际支持的值可以使用下面的代码查询：

```
GLint maxVertexAttribs, maxVertexUniforms, maxVaryings;
GLint maxVertexTextureUnits, maxCombinedTextureUnits;
glGetIntegerv(GL_MAX_VERTEX_ATTRIBS, &maxVertexAttribs);
glGetIntegerv(GL_MAX_VERTEX_UNIFORM_VECTORS, &maxVertexUniforms);
glGetIntegerv(GL_MAX_VARYING_VECTORS, &maxVaryings);
glGetIntegerv(GL_MAX_VERTEX_TEXTURE_IMAGE_UNITS,
              &maxVertexTextureUnits);
glGetIntegerv(GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS,
              &maxCombinedTextureUnits);
```

精度控制

我们做一个简短的精度控制的概括。精度控制在第 5 章被使用在指示浮点和整形变量值，指定的精度可以是低、中和高。声明精度质量的例子如下：

```
highp vec4          position;
varying lowp vec4   color;
mediump float       specularExp;
```

另外，也有默认的物体精度质量。如果变量声明时没有指明精度质量，对这种变量类型有默认的精度质量。默认的精度质量在顶点着色器和片段着色器的头部使用下面的语法指定：

```
precision highp float;
precision mediump int;
```

对所有的浮点变量指定的精度作为所有基于浮点值的默认精度。对所有的整型变量指定的精度作为所有基于整型值的默认精度。在顶点着色器里，如果没有指定精度，对整型和浮点默认的精度是高。

在顶点着色器里的操作，默认的精度大多数情况下是高精度。使用位置变换矩阵、法线转换矩阵和贴图坐标转换矩阵或者产生贴图坐标都是用高质量。颜色计算和产生光照方程大部分使用中等质量。这依靠于颜色计算执行的种类、范围、操作要求的精度。我们一般认为高精度是顶点着色器默认的大多数的操作要求，使用默认的精度的例子如下。

ES 2.0 顶点着色器限制

这部分讨论 OpenGL ES 2.0 顶点着色器的着色器语言的限制。这些限制将有助于移植顶点着色器能在大多数的 OpenGL ES 2.0 编译器里执行。

顶点着色器长度

没有方法能够查询 OpenGL ES 2.0 编译器的顶点着色器支持的最多的指令数。因此无法确定一个编译器支持的顶点着色器指令数是多或少。但如果指令超过了顶点着色器允许的最大指令数，着色器源码将编译失败。

OpenGL ES 工作组意识到不能够查询最大数和着色器里的实际 uniform 数是严重的问题，将让开发者工作变得困难。计划去提供一套顶点着色器（片段着色器）帮助确认指令的复杂度和 uniform 的使用。另外这个着色器将是 OpenGL ES 2.0 一致性测试的一部分，这意味着所有 OpenGL ES 2.0 通过一致性测试的编译器将有运行着色器的能力。

临时变量

临时变量是指在函数内部声明的变量或者存储中间变量的变量。OpenGL ES 着色器语言是一种高级语言，没有办法指定 OpenGL ES 2.0 必须支持的临时变量数目。所以一个顶点着色器可能在一个上面运行，可能不能在所有的 ES 2.0 编译器上兼容。

流控制

OpenGL ES 2.0 要求顶点着色器支持循环，即使它们没有展开。例如你能有一个循环索引从 0 到 1023 的循环。这在着色器编译时将不会展开，展开后的代码对 ES 2.0 编译器工具将太大。

顶点着色器使用循环将有下面的限制：

一个循环只有一个循环索引

循环索引必须被一个整形常量表达初始化。

表达条件必须是下面中的一个

loop_indx < constant_expression

loop_indx <= constant_expression

loop_indx > constant_expression

loop_indx >= constant_expression

loop_indx != constant_expression

loop_indx == constant_expression

循环状态必须使用下面的表达式修改

loop_index--

loop_index++

loop_index -= constant_expression

loop_index += constant_expression

循环索引值必须传递一个只读的参数到函数内开始循环。（例如循环索引被用作有限定的参数声明）。

一个有效的循环结构如下：

```
const int numLights = 4;
int i, j;
for (i=0; i<numLights; i++)
{
    …
}
for (j=4; j>0; j--)
{
    …
    foo(j);    // argument to function foo that takes j
                  // is declared with the in qualifier.
}
```

错误的循环结构如下：

```
uniform int numLights;
int i;
for (i=0; i<numLights; i++)    // conditional expression is
                                    // not constant
{
    …
}
for (i=0; i<8; i++)
{
    i = foo();      // return value of foo() cannot be
                       // assigned to loop index i
}
for (j=4; j>0;)
{
    …
    j--;      // loop index j cannot be modified
                 // inside for loop
}
```

while 和 do-while 循环，尽管被 OpenGL ES 2.0 着色器语言说明书指定，但不是必须地要求，一个 OpenGL ES 2.0 工具可以不支持它。

条件语句

下面的条件语句没有任何限制的被所有编译器支持

```
if(bool_expression)
{
    …
}
if(bool_expression)
{
    …
}
else
```

```
{
    …
}
```

bool_expression 必须是一个标量布尔值

GPU 一般并行执行多个顶点的顶点着色器或执行多个片段的片段着色器。并行执行的顶点数或者片段数依靠 GPU 执行的目标。if 和 if-else 的条件状态 bool_expression 在并行处理时可能有不同的值。减少 GPU 并行处理的顶点和片段数目将降低性能。为更好的性能我们推荐对顶点和片段使用的相同的 bool_expression 值并行计算。这是常量表达式被使用的情况。

数组索引

uniform 的数组索引（除了采样）都被支持。数字索引能够是常量、uniform 或计算值。采样仅仅被整型常量表达式索引支持。整型常量表达式是一个文字值（例如 4），一个恒定的整型变量（例如 const int sampler_indx = 3），或者常量表达式（例如 3 + sampler_indx）。

属性矩阵和向量能使用整型常量表达式索引。使用整型非常量表达式将不能托管。然而无论怎么说这是一个有用的特征。下面的代码允许一个顶点着色器执行顶点皮肤，a_matrixweights 是存储在矩阵中的顶点属性，它有 4 个元素。

```
attribute   vec4 a_matrixweights; //   matrix   weights
attribute  vec4 a_matrixindices;   // matrix palette indices
int i;
for (i=0; i<=3; i++)
{
    float     m_wt = a_matrixweights[i];
    int        m_indx = int(a_matrixindices[i]) * 3;
    …
}
```

突出显示的 a_matrixweights[i]和 a_matrixindices[i]是不要求被支持的，可能编译失败。

注意：矩阵、矢量、变化量、变量或者变量数组的常量索引规则和上面已经描述的属性规则相同。

计算顶点着色器使用的 uniform 数目

gl_MaxVertexUniformVectors 描述顶点着色器里使用的 uniform 的最大数目。最小值是 128 个 vec4 输入。uniform 用来存储下面的变量：

使用 uniform 限定的变量声明

常数变量

文字值

执行器特殊常量

在顶点着色器里 uniform 的数目和带常数限制的变量声明、文字值、执行工具的特殊的常量的包装规则必须适合第 5 章中 gl_MaxVertexUniformVectors 的描述。如果不适合，编译器将编译失败。对开发者来说，应用包装规则决定 uniform 存储需要存储的 uniform、恒定常量和文字值的数目是可能的。决定编译器特殊常量的数目是不可能的，因为这个值不但从编译器到编译器会发生变化，而且在顶点着色器的内建语言函数中使用时也会发生变化。典型的，编译器特殊常量被要求编译为超越函数去使用。

至于文字值，OpenGL ES 2.0 着色器语言没有提供常数传用说明。这意味着相同的文字值的多个实例将被多次计算。在顶点着色器里使用 0.0 或 1.0 这种文字值是很容易理解的，但我们推荐尽量避免这种使用。替代使用文字值为合适的整型变量。这将避免相同的文字值

被多次计数。如果顶点 uniform 超过了编译器支持的范围，顶点着色器将编译失败。

下面的代码描述顶点着色器的逐像素转换两个贴图坐标的一个片段。

```
#define NUM_TEXTURES      2
uniform mat4    tex_matrix[NUM_TEXTURES];              // texture matrices
uniform bool    enable_tex[NUM_TEXTURES];              // texture enables
uniform bool    enable_tex_matrix[NUM_TEXTURES]; // texture matrix
                                                       // enables
attribute vec4    a_texcoord0;    // available if enable_tex[0] is true
attribute vec4    a_texcoord1;    // available if enable_tex[1] is true
varying vec4        v_texcoord[NUM_TEXTURES];
v_texcoord[0] = vec4(0.0, 0.0, 0.0, 1.0);
// is texture 0 enabled
if (enable_tex[0])
{
    // is texture matrix 0 enabled
    if(enable_tex_matrix[0])
        v_texcoord[0] = tex_matrix[0] * a_texcoord0;
    else
        v_texcoord[0] = a_texcoord0;
}
v_texcoord[1] = vec4(0.0, 0.0, 0.0, 1.0);
// is texture 1 enabled
if (enable_tex[1])
{
    // is texture matrix 1 enabled
    if(enable_tex_matrix[1])
        v_texcoord[1] = tex_matrix[1] * a_texcoord1;
    else
        v_texcoord[1] = a_texcoord1;
}
```

上面的代码可能导致两个文字值是 0, 1, 0.0，1.0，不利于存储。为保证这些文字值仅仅保存一次，顶点着色器代码应该这样写：

```
#define NUM_TEXTURES      2
const int    c_zero = 0;
const int    c_one    = 1;
uniform mat4    tex_matrix[NUM_TEXTURES];              // texture matrices
uniform bool    enable_tex[NUM_TEXTURES];              // texture enables
uniform bool    enable_tex_matrix[NUM_TEXTURES]; // texture matrix
                                                       // enables
attribute vec4    a_texcoord0;    // available if enable_tex[0] is true
attribute vec4    a_texcoord1;    // available if enable_tex[1] is true
varying vec4        v_texcoord[NUM_TEXTURES];
v_texcoord[c_zero] = vec4(float(c_zero), float(c_zero),
                                    float(c_zero), float(c_one));
```

```
    // is texture 0 enabled
    if(enable_tex[c_zero])
    {
        // is texture matrix 0 enabled
        if(enable_tex_matrix[c_zero])
            v_texcoord[c_zero] = tex_matrix[c_zero] * a_texcoord0;
        else
            v_texcoord[c_zero] = a_texcoord0;
    }
    v_texcoord[c_one] = vec4(float(c_zero), float(c_zero),
                                        float(c_zero), float(c_one));
    // is texture 1 enabled
    if(enable_tex[c_one])
    {
        // is texture matrix 1 enabled
        if(enable_tex_matrix[c_one])
            v_texcoord[c_one] = tex_matrix[c_one] * a_texcoord1;
        else
            v_texcoord[c_one] = a_texcoord1;
    }
```

希望这章有助于理解 OpenGL ES 2.0 着色器语言的限制，怎样写着色器将在大多数的 OpenGL ES 2.0 编译工具上编译运行。

顶点着色器例子

现在提供几个例子展示怎样在顶点着色器执行下面的特征。

使用矩阵转换顶点位置

光照计算产生每像素的散射和镜面反射颜色

贴图坐标产生

顶点皮肤

这些特征代表了 OpenGL ES 2.0 应用将在着色器执行的典型行为。

一个简单的顶点着色器

例 8-1 描述使用 OpenGL ES 2.0 着色器语言的简单的着色器。它使用一个位置值，关联一个颜色数据做输入和属性，使用 4 ×4 的矩阵转换位置，输出转换后的位置和颜色。

例 8-1 一个简单的顶点着色器

```
// uniforms used by the vertex shader
uniform mat4       u_mvp_matrix; // matrix to convert P from
                                        // model space to clip space.
// attributes input to the vertex shader
attribute vec4    a_position;     // input position value
attribute vec4    a_color;          // input vertex color
// varying variables – input to the fragment shader
varying vec4       v_color;          // output vertex color
void
main()
{
```

```
    v_color = a_color;
    gl_Position = u_mvp_matrix * a_position;
}
```

顶点位置被转换，基元类型被设置后，光栅化阶段将光栅化基元成为片段。对每个片段，插补后的 v_color 被计算，作为片段着色器的输入。

顶点着色器的光照

这章我们看一些计算平行光源、点光源和聚光灯的光照方程的例子。在这章的顶点着色器使用 OpenGL ES 1.1 光照模型去计算平行和点光源。这个光照例子，观察者被假定在无穷远处。

平行光是光源相对于场景中的物体在无穷远处。平行光的例子是太阳。当光源在无穷远处时，从光源发出的光线是平行的。光方向矢量是常量，不需要每个顶点计算。图 8-3 描述了对平行光需要计算的光方程。P_{eye} 是观察者位置，P_{light} 是光的位置（P_{light} .w = 0），N 是法线，H 是半平面矢量，因为 P_{light} . w = 0，光线方向矢量将是 P_{light} . xyz，半平面方向 H 使用|| VP_{light} + VP_{eye} ||计算。因为光源和观察者在无穷远处，半平面矢量 H=|| P_{light} . xyz + (0, 0, 1) ||。

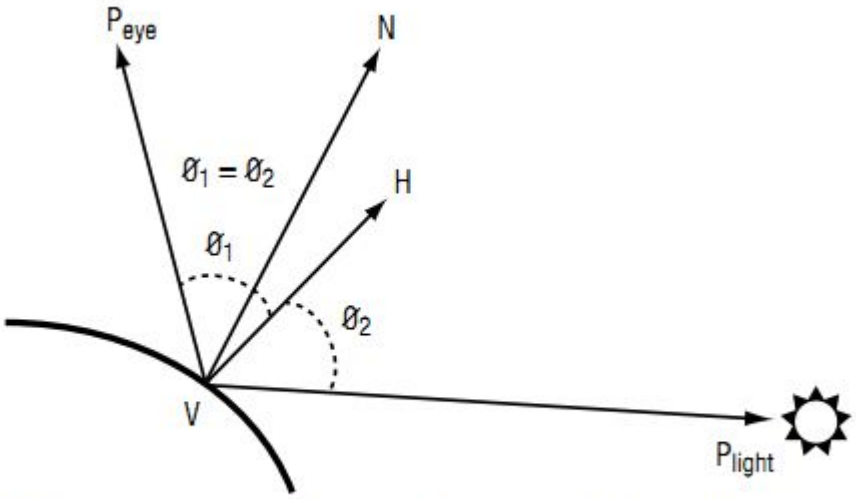


Figure 8-3 Geometric Factors in Computing Lighting Equation for a Directional Light

例 8-2 描述了计算代码是计算方向性光的方程。方向性光源使用 directional_light 结构描述，它的成员包括：

direction—眼睛空间标准化的光线方向

halfplane—归一化的半平面矢量 H，对方向性光源，应该预编译，不能变换。

ambient_color—光源环境的颜色

diffuse_color—光源散射的颜色

specular_color—光源镜面反射的颜色

计算顶点散射和镜面反射的颜色的材料属性决定于 material_properties 结构，它的元素包括：

- ambient_color—材料的环境光颜色
- diffuse_color—材料的漫射光颜色
- specular_color—材料的镜面反射光颜色
- specular_exponent—这个反射指数描述材料发光，使用去控制反射光的光照情形

例 8-2 方向光

```
struct directional_light {
    vec3    direction;          // normalized light direction in eye space
    vec3    halfplane;          // normalized half-plane vector
    vec4    ambient_color;
    vec4    diffuse_color;
    vec4    specular_color;
};
struct material_properties {
    vec4     ambient_color;
    vec4     diffuse_color;
    vec4     specular_color;
    float    specular_exponent;
};
const float                         c_zero = 0.0;
const float                         c_one = 1.0;
uniform material_properties    material;
uniform directional_light       light;
// normal has been transformed into eye space and is a normalized
// value returns the computed color.
void
directional_light(vec3 normal)
{
    vec4     computed_color = vec4(c_zero, c_zero, c_zero, c_zero);
    float    ndotl;    // dot product of normal & light direction
    float    ndoth;    // dot product of normal & half-plane vector
    ndotl = max(c_zero, dot(normal, light.direction));
    ndoth = max(c_zero, dot(normal, light.halfplane));
    computed_color += (light.ambient_color * material.ambient_color);
    computed_color += (ndotl * light.diffuse_color
                                * material.diffuse_color);
    if (ndoth > c_zero)
    {
        computed_color += (pow(ndoth, material.specular_exponent) *
                                    material.specular_color *
                                    light.specular_color);
    }
    return computed_color;
}
```

例 8-2 中顶点着色器直射光源码混合每个顶点的散射和镜面反射颜色成单一颜色（即 computed_color）。另一种方法是计算每个顶点的散射光和镜面反射光颜色，把它们作为独立的变量输入到片段着色器。

注意：例 8-2，我们用光的颜色乘了物质材料颜色（环境、散射、镜面反射）。如果我们仅仅计算一种光，使用光方程是比较好的选择。如果是计算多种光的光方程，应该计算环境、散射和镜面反射等各个光的值，然后通过乘物质材料的环境、散射和镜面反射进行合适的计

算，在求和去产生每个顶点颜色。

点光源是从空间一个点向所有方向发射光线的光源。点光源被设置为(x, y, z, w)，其中 w ≠ 0。点光源向所有方向发射光线，但它的强度依据光源到物体之间的距离而衰减。衰减计算使用下面的方程：

距离衰减=$1 / (K_0 + K_1 \times \| VP_{light} \| + K_2 \times \| VP_{light} \|^2)$

K_0 、 K_1 和 K_2 是常量、线性和二次衰减因子。

聚光灯是圆锥形的具有位置还有方向的光源，它从一个位置（P_{light}）沿一个方向发出（$spot_{direction}$）。图 8-4 描述了聚光灯光源需要计算的光方程。

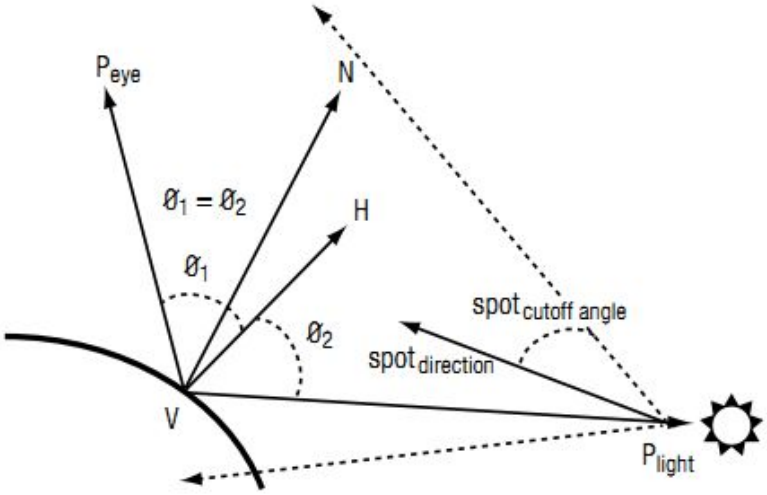


Figure 8-4 Geometric Factors in Computing Lighting Equation for a Spot Light

反射光的强度基于到圆锥体中心的角度即现场截止因子而衰减。到圆锥中心的角度作为 VP_{light} 和 $spot_{direction}$ 的标量积被计算。$spot_{direction}$ 这个因子在聚光灯的位置是 1.0，随弧度 $spot_{cutoff\ angle}$ 以指数规律减小到 0.0。

例 8-3 描述了计算聚光灯的光照方程的顶点着色器源码。聚光灯的性能使用 spot_light 结构体描述，它包含这些元素：

- direction—眼睛空间的光线方向
- ambient_color—光线的环境颜色
- diffuse_color—光线的漫射颜色
- specular_color—光线的镜面反射颜色
- attenuation_factors—距离衰变因子 K0, K1, 和 K2.
- compute_distance_attenuation—这个布尔值决定是否距离衰减是否必须被计算
- spot_direction—标准化的点距离矢量
- spot_exponent—使用去计算点截止因子的聚光灯指数
- spot_cutoff_angle—聚光灯的截止角度

例 8-3 聚光灯光照

```
struct spot_light {
    vec4    position;                   // light position in eye space
    vec4    ambient_color;
    vec4    diffuse_color;
    vec4    specular_color;
    vec3    spot_direction;         // normalized spot direction
```

```
    vec3       attenuation_factors; // attenuation factors K0, K1, K2
    bool       compute_distance_attenuation;
    float      spot_exponent;             // spotlight exponent term
    float      spot_cutoff_angle;      // spot cutoff angle in degrees
};
struct material_properties {
    vec4       ambient_color;
    vec4       diffuse_color;
    vec4       specular_color;
    float      specular_exponent;
};
const float                          c_zero = 0.0;
const float                          c_one = 1.0;
uniform material_properties    material;
uniform spot_light                   light;
// normal and position are normal and position values in eye space.
// normal is a normalized vector.
// returns the computed color.

vec4
spot_light(vec3 normal, vec4 position)
{
    vec4       computed_color = vec4(c_zero, c_zero, c_zero, c_zero);
    vec3       lightdir;
    vec3       halfplane;
    float      ndotl, ndoth;
    float      att_factor;
    att_factor = c_one;
    // we assume "w" values for light position and
    // vertex position are the same
    lightdir = light.position.xyz - position.xyz;
    // compute distance attenuation
    if(light.compute_distance_attenuation)
    {
        vec3       att_dist;
        att_dist.x = c_one;
        att_dist.z = dot(lightdir, lightdir);
        att_dist.y = sqrt(att_dist.z);
        att_factor = c_one / dot(att_dist, light.attenuation_factors);
    }
    // normalize the light direction vector
    lightdir = normalize(lightdir);
    // compute spot cutoff factor
    if(light.spot_cutoff_angle < 180.0)
```

```
    {
        float    spot_factor = dot(-lightdir, light.spot_direction);
        if(spot_factor >= cos(radians(light.spot_cutoff_angle)))
            spot_factor = pow(spot_factor, light.spot_exponent);
        else
            spot_factor = c_zero;
        // compute combined distance & spot attenuation factor
        att_factor *= spot_factor;
    }
    if(att_factor > c_zero)
    {
        // process lighting equation --> compute the light color
        computed_color += (light.ambient_color *
                                material.ambient_color);
        ndotl = max(c_zero, dot(normal, lightdir));
        computed_color += (ndotl * light.diffuse_color *
                                material.diffuse_color);
        halfplane = normalize(lightdir + vec3(c_zero, c_zero, c_one));
        ndoth = dot(normal, halfplane);
        if (ndoth > c_zero)
        {
            computed_color += (pow(ndoth, material.specular_exponent)*
                                    material.specular_color *
                                    light.specular_color);
        }
        // multiply color with computed attenuation
        computed_color *= att_factor;
    }
    return computed_color;
}
```

产生贴图坐标

我们看两个顶点着色器产生贴图坐标的例子。这两个例子渲染了场景中有光泽的物体，通过产生反射矢量，然后使用这个矢量去计算匹配经度或纬度（也称球形匹配）的贴图坐标或者匹配立方体（代表 6 个观察面或表面，假定在发光体的中间只有一个捕捉环境反射的观察面）贴图。OpenGL 2.0 使用 GL_SPHERE_MAP 和 GL_REFLECTION_MAP 描述贴图坐标产生的模型。GL_SPHERE_MAP 模型使用反射矢量去计算 2D 贴图坐标查找 2D 贴图的匹配。GL_REFLECTION_MAP 模型使用反射矢量去计算 3D 贴图坐标查找立方体贴图的匹配。例 8-4 和 8-5 描述了计算有光泽物体的反射图像的着色器源码，它们产生的贴图坐标将被片段着色器合理的使用。

Example 8-4 匹配球体的贴图坐标产生

```
// position is the normalized position coordinate in eye space
// normal is the normalized normal coordinate in eye space
// returns a vec2 texture coordinate
vec2
```

```
sphere_map(vec3 position, vec3 normal)
{
    reflection = reflect(position, normal);
    m = 2.0 * sqrt(reflection.x * reflection.x +
                   reflection.y * reflection.y +
                   (reflection.z + 1.0) * (reflection.z + 1.0));
    return vec2((reflection.x / m + 0.5), (reflection.y / m + 0.5));
}
```

Example 8-5 匹配立方体的贴图坐标产生

```
// position is the normalized position coordinate in eye space
// normal is the normalized normal coordinate in eye space
// returns the reflection vector as a vec3 texture coordinate
vec3
cube_map(vec3 position, vec3 normal)
{
    return reflect(position, normal);
}
```

反射矢量将在片段着色器里作为合适的立方体匹配的贴图坐标使用。

顶点拟合

顶点拟合是普遍使用的让多边形更光滑的技术。它通过对每个顶点应用合适因子的转换矩阵来实现。使用拟合的多重因子被存储到矩阵调色板里。被用作顶点拟合的每个顶点的矩阵索引被用来替代矩阵调色板里的对应的矩阵因子。顶点拟合对 3D 游戏里的模型是普遍使用的，目的是为了确保不使用更多的几何图形而让模型看起来更光滑和具有真实感（尽可能的）。用来拟合的矩阵数目一般是两个或四个。

顶点拟合使用下面的方程运算：

$P' = \Sigma w_i \times M_i \times P$

$N' = \Sigma w_i \times M_i^{-1T} \times N$

$\Sigma w_i = 1, i = 1 \text{ to } n$

这里

n 是转换顶点的矩阵数目

P 是顶点位置

P′ 是转换（拟合）位置

N 是顶点法线

N′ 是转换（拟合）法线

M_i 是索引值为 i^{th} 的联系每个顶点的矩阵，计算如下：

M = matrix_palette [matrix_index[i]]

n 是指示顶点的矩阵索引值

M_i^{-1T} 是矩阵 M_i 的转置矩阵

w_i 是联系矩阵的权重

我们讨论使用 32 个矩阵的矩阵模板和每个顶点四个矩阵去产生拟合后的顶点。32 个矩阵的矩阵模板是很普遍的。矩阵模板里的矩阵一般是 4 × 3 列序矩阵（四个 vec3）。如果矩阵以列序存储，它将有 128 个 uniform 输入，每个输入有三个元素存储成一行。OpenGL ES 2.0 支持的 gl_MaxVertexUniformVectors 最小值是 128 个 vec4。这意味着我们在 128 个 vec4 中仅仅有四个行能被使用。浮点的行仅能被存储在声明为类型是浮点的区域（像 uniform 包

装规则）。没有空间去存储 vec2, vec3, 或 vec4 uniform。在模板里每个矩阵使用三个 vec4 entries，使用行主序存储矩阵是更有效的。如果我们这样做我们仅仅使用了 96 个 vec4's uniform 存储空间，还剩 32 个 vec4 输入口能被用于其它用途。注意我们没有足够的空间去存储计算拟合法线的转换矩阵的逆矩阵。这在大部分用来转换顶点位置和法线的正交矩阵的情况下不是问题。

例 8-6 描述计算拟合法线和位置的顶点着色器源码。假设调色板有 32 个矩阵，矩阵以行主序存储。矩阵假定是正交的（被使用来转换位置和法线的相似的矩阵），有四个矩阵被使用转换每个顶点。

例 8-6 不检查矩阵权重等于 0 的顶点拟合着色器

```
#define NUM_MATRICES    32    // 32 matrices in matrix palette
const int           c_zero    = 0;
const int           c_one     = 1;
const int           c_two     = 2;
const int           c_three = 3;
// store 32 4 x 3 matrices as an array of floats representing
// each matrix in row-major order i.e. 3 vec4s
uniform vec4        matrix_palette[NUM_MATRICES * 3];
// vertex position and normal attributes
attribute vec4    a_position
attribute vec3    a_normal;
// matrix weights - 4 entries / vertex
attribute vec4    a_matrixweights;
// matrix palette indices
attribute vec4    a_matrixindices;
void
skin_position(in vec4 position, float m_wt, int m_indx,
                                        out vec4 skinned_position)
{
    vec4        tmp;
    tmp.x = dot(position, matrix_palette[m_indx]);
    tmp.y = dot(position, matrix_palette[m_indx + c_one]);
    tmp.z = dot(position, matrix_palette[m_indx + c_two]);
    tmp.w = position.w;

    skinned_position += m_wt * tmp;
}
void
skin_normal(in vec3 normal, float m_wt, int m_indx,
                                        out vec3 skinned_normal)
{
    vec3        tmp;
    tmp.x = dot(normal, matrix_palette[m_indx].xyz);
    tmp.y = dot(normal, matrix_palette[m_indx + c_one].xyz);
    tmp.z = dot(normal, matrix_palette[m_indx + c_two].xyz);
```

```
    skinned_position += m_wt * tmp;
}
void
do_skinning(in vec4 position, in vec3 normal,
                    out vec4 skinned_position, out vec4 skinned_normal)
{
    skinned_position = vec4(float(c_zero));
    skinned_normal = vec3(float(c_zero));
    // transform position and normal to eye space using matrix
    // palette with four matrices used to transform a vertex
    m_wt = a_matrixweights[0];
    m_indx = int(a_matrixindices[0]) * c_three;
    skin_position(position, m_wt, m_indx, skinned_position);
    skin_normal(normal, m_wt, m_indx, skinned_normal);
    m_wt = a_matrixweights[1];
    m_indx = int(a_matrixindices[1]) * c_three;
    skin_position(position, m_wt, m_indx, skinned_position);
    skin_normal(normal, m_wt, m_indx, skinned_normal);
    m_wt = a_matrixweights[2];
    m_indx = int(a_matrixindices[2]) * c_three;
    skin_position(position, m_wt, m_indx, skinned_position);
    skin_normal(normal, m_wt, m_indx, skinned_normal);
    m_wt = a_matrixweights[3];
    m_indx = int(a_matrixindices[3]) * c_three;
    skin_position(position, m_wt, m_indx, skinned_position);
    skin_normal(normal, m_wt, m_indx, skinned_normal);
}
```

例 8-6 中，顶点拟合着色器使用四个矩阵和合适的矩阵权重转换产生一个拟合顶点。矩阵的权重中的很多值很可能或者非常普通都是 0。例 8-6 中，顶点不管它们的权重，使用四个矩阵转换。在调用 skin_position 和 skin_normal 前，检查矩阵权重是否为 0 可能更好。例 8-7，描述使用机制转换前检查权重是否为 0 的顶点拟合着色器。

例 8-7 检查矩阵权重是否等于 0 的顶点拟合着色器

```
void
do_skinning(in vec4 position, in vec3 normal,
                    out vec4 skinned_position, out vec4 skinned_normal)
{
    skinned_position = vec4(float(c_zero));
    skinned_normal = vec3(float(c_zero));
    // transform position and normal to eye space using matrix
    // palette with four matrices used to transform a vertex
    m_wt = a_matrixweights[0];
    if(m_wt > 0.0)
    {
        m_indx = int(a_matrixindices[0]) * c_three;
```

```
        skin_position(position, m_wt, m_indx, skinned_position);
        skin_normal(normal, m_wt, m_indx, skinned_normal);
    }
    m_wt = a_matrixweights[1];
    if(m_wt > 0.0)
    {
        m_indx = int(a_matrixindices[1]) * c_three;
        skin_position(position, m_wt, m_indx, skinned_position);
        skin_normal(normal, m_wt, m_indx, skinned_normal);
    }
    m_wt = a_matrixweights[2];
    if(m_wt > 0.0)
    {
        m_indx = int(a_matrixindices[2]) * c_three;
        skin_position(position, m_wt, m_indx, skinned_position);
        skin_normal(normal, m_wt, m_indx, skinned_normal);
    }
    m_wt = a_matrixweights[3];
    if(m_wt > 0.0)
    {
        m_indx = int(a_matrixindices[3]) * c_three;
        skin_position(position, m_wt, m_indx, skinned_position);
        skin_normal(normal, m_wt, m_indx, skinned_normal);
    }
}
```

容易理解，例 8-7 中的顶点拟合着色器比例 8-6 中的顶点拟合着色器执行的更好。但这未必正确，答案还依靠 GPU。这是因为条件表达式 if (m_wt > 0.0)中，m_wt 是一个动态值，在 GPU 并行执行时对正在执行的顶点是变化的。当顶点被并行执行时，使用了分流控制，m_wt 值有不同的值，执行必须等待。如果 GPU 不能提高分流控制的效率，例 8-7 中的顶点拟合着色器效率可能低于例 8-6 中的顶点拟合着色器。因此，应用必须在应用初始化阶段测试 GPU 分流控制的执行情况，决定那种着色器将被使用。

我们希望这些例子让你对顶点着色器懂得更多，怎么写他们，怎么使用它们有更高的效率。

OpenGL ES 1.1 顶点管道和 OpenGL ES 2.0 顶点着色器

我们谈到 OpenGL ES 1.1 的固定行为的管道是没有顶点拟合的。这意味着编写能够被 ES2.0 运行的 OpenGL ES 1.1 着色器程序是有意思的事。

OpenGL ES 1.1 的顶点管道着色器应用有下面的固定行为

如果需要（光照一般需要），转换法线和位置到眼睛空间。标准化法线。

计算最多 8 个方向的 OpenGL ES 1.1 顶点光照方程、点或者聚光灯两边的光以及每个顶点的材料颜色。

为每个顶点的转换最多两个贴图坐标。

计算传递到片段着色器的雾因子。片段着色器使用雾因子在雾颜色和顶点颜色之间插补。

计算每个顶点的用户截面系数，仅仅支持一个用户截面。

转换位置到剪切空间。

例 8-8 是 OpenGL ES 1.1 固定功能行为的管道顶点着色器应用。

例 8-8 固定功能行为的顶点管道

```
//*********************************************************************
//
// OpenGL ES 2.0 vertex shader that implements the following
// OpenGL ES 1.1 fixed function pipeline
//
// - compute lighting equation for up to eight directional/point/
// - spot lights
// - transform position to clip coordinates
// - texture coordinate transforms for up to two texture coordinates
// - compute fog factor
// - compute user clip plane dot product (stored as v_ucp_factor)
//
//*********************************************************************
#define NUM_TEXTURES                    2
#define GLI_FOG_MODE_LINEAR             0
#define GLI_FOG_MODE_EXP                1
#define GLI_FOG_MODE_EXP2               2
struct light {
    vec4      position;    // light position for a point/spot light or
                           // normalized dir. for a directional light
    vec4      ambient_color;
    vec4      diffuse_color;
    vec4      specular_color;
    vec3      spot_direction;
    vec3      attenuation_factors;
    float     spot_exponent;
    float     spot_cutoff_angle;
    bool      compute_distance_attenuation;
};
struct material {
    vec4      ambient_color;
    vec4      diffuse_color;
    vec4      specular_color;
    vec4      emissive_color;
    float     specular_exponent;
};
const float          c_zero = 0.0;
const float          c_one = 1.0;
const int            indx_zero = 0;
const int            indx_one = 1;
uniform mat4         mvp_matrix;     // combined model-view +
```

```
                                        // projection matrix
uniform mat4          modelview_matrix;         // model view matrix
uniform mat3          inv_modelview_matrix; // inverse model-view
                                                // matrix used
// to transform normal
uniform mat4          tex_matrix[NUM_TEXTURES]; // texture matrices
uniform bool          enable_tex[NUM_TEXTURES]; // texture enables
uniform bool          enable_tex_matrix[NUM_TEXTURES]; // texture matrix
                                                       // enables
uniform material    material_state;
uniform vec4          ambient_scene_color;
uniform light         light_state[8];
uniform bool          light_enable_state[8];    // booleans to indicate
                                                // which of eight
                                                // lights are enabled
uniform int           num_lights;// number of lights enabled = sum of
                                 // light_enable_state bools set to TRUE
uniform bool          enable_lighting;              // is lighting enabled
uniform bool          light_model_two_sided;    // is two-sided lighting
                                                // enabled
uniform bool          enable_color_material;    // is color material
                                                // enabled
uniform bool          enable_fog;                   // is fog enabled
uniform float         fog_density;
uniform float         fog_start, fog_end;
uniform int           fog_mode;                     // fog mode - linear, exp,
                                                // or exp2
uniform bool          xform_eye_p;      // xform_eye_p is set if we need
                                        // Peye for user clip plane,
                                        // lighting, or fog
uniform bool          rescale_normal;         // is rescale normal enabled
uniform bool          normalize_normal;      // is normalize normal enabled
uniform float         rescale_normal_factor; // rescale normal factor if
                                             // glEnable(GL_RESCALE_NORMAL)
uniform vec4          ucp_eqn;            // user clip plane equation –
                                        // - one user clip plane specified
uniform bool          enable_ucp;       // is user clip plane enabled
//*************************************************************
// vertex attributes - not all of them may be passed in
//*************************************************************
attribute vec4        a_position; // this attribute is always specified
attribute vec4        a_texcoord0;// available if enable_tex[0] is true
attribute vec4        a_texcoord1;// available if enable_tex[1] is true
attribute vec4        a_color;        // available if !enable_lighting or
```

```
                                    // (enable_lighting && enable_color_material)
attribute vec3        a_normal;     // available if xform_normal is set
                                          // (required for lighting)
//*****************************************************
// varying variables output by the vertex shader
//*****************************************************
varying vec4             v_texcoord[NUM_TEXTURES];
varying vec4             v_front_color;
varying vec4             v_back_color;
varying float            v_fog_factor;
varying float            v_ucp_factor;
//*****************************************************
// temporary variables used by the vertex shader
//*****************************************************
vec4                        p_eye;
vec3                        n;
vec4                        mat_ambient_color;
vec4                        mat_diffuse_color;
vec4
lighting_equation(int i)
{
    vec4       computed_color = vec4(c_zero, c_zero, c_zero, c_zero);
    vec3       h_vec;
    float      ndotl, ndoth;
    float      att_factor;
    att_factor = c_one;
    if(light_state[i].position.w != c_zero)
    {
        float      spot_factor;
        vec3       att_dist;
        vec3       VPpli;
        // this is a point or spot light
        // we assume "w" values for PPli and V are the same
        VPpli = light_state[i].position.xyz - p_eye.xyz;
        if(light_state[i].compute_distance_attenuation)
        {
            // compute distance attenuation
            att_dist.x = c_one;
            att_dist.z = dot(VPpli, VPpli);
            att_dist.y = sqrt(att_dist.z);
            att_factor = c_one / dot(att_dist,
                light_state[i].attenuation_factors);
        }
        VPpli = normalize(VPpli);
```

```
        if(light_state[i].spot_cutoff_angle < 180.0)
        {
            // compute spot factor
            spot_factor = dot(-VPpli, light_state[i].spot_direction);
            if(spot_factor >= cos(radians(
                                        light_state[i].spot_cutoff_angle)))
                spot_factor = pow(spot_factor,
                                       light_state[i].spot_exponent);
            else
                spot_factor = c_zero;
            att_factor *= spot_factor;
        }
    }
    else
    {
        // directional light
        VPpli = light_state[i].position.xyz;
    }
    if(att_factor > c_zero)
    {
        // process lighting equation --> compute the light color
        computed_color += (light_state[i].ambient_color *
                                mat_ambient_color);
        ndotl = max(c_zero, dot(n, VPpli));
        computed_color += (ndotl * light_state[i].diffuse_color *
                                mat_diffuse_color);
        h_vec = normalize(VPpli + vec3(c_zero, c_zero, c_one));
        ndoth = dot(n, h_vec);
        if (ndoth > c_zero)
        {
            computed_color += (pow(ndoth,
                                   material_state.specular_exponent) *
                                   material_state.specular_color *
                                   light_state[i].specular_color);
         }
         computed_color *= att_factor; // multiply color with
                                            // computed attenuation factor
                                            // * computed spot factor
    }
    return computed_color;
}
float
compute_fog()
{
```

```
    float      f;
    // use eye Z as approximation
    if(fog_mode == GLI_FOG_MODE_LINEAR)
    {
        f = (fog_end - p_eye.z) / (fog_end - fog_start);
    }
    else if(fog_mode == GLI_FOG_MODE_EXP)
    {
        f = exp(-(p_eye.z * fog_density));
    }
    else
    {
        f = (p_eye.z * fog_density);
        f = exp(-(f * f));
    }
    f = clamp(f, c_zero, c_one);
    return f;
}
vec4
do_lighting()
{
    vec4        vtx_color;
    int         i, j;
    vtx_color = material_state.emissive_color +
                        (mat_ambient_color * ambient_scene_color);
    j = (int)c_zero;
    for (i=(int)c_zero; i<8; i++)
    {
        if(j >= num_lights)
            break;
        if (light_enable_state[i])
        {
            j++;
            vtx_color += lighting_equation(i);
        }
    }
    vtx_color.a = mat_diffuse_color.a;
    return vtx_color;
}
void
main(void)
{
    int         i, j;
    // do we need to transform P
```

```
if(xform_eye_p)
    p_eye = modelview_matrix * a_position;
if(enable_lighting)
{
    n = inv_modelview_matrix * a_normal;
    if(rescale_normal)
        n = rescale_normal_factor * n;
    if (normalize_normal)
            n = normalize(n);
    mat_ambient_color = enable_color_material ? a_color
                                              : material_state.ambient_color;
    mat_diffuse_color = enable_color_material ? a_color
                                              : material_state.diffuse_color;
    v_front_color = do_lighting();
    v_back_color = v_front_color;
    // do 2-sided lighting
    if(light_model_two_sided)
    {
        n = -n;
        v_back_color = do_lighting();
    }
}
else
{
    // set the default output color to be the per-vertex /
    // per-primitive color
    v_front_color = a_color;
    v_back_color = a_color;
}
// do texture xforms
v_texcoord[indx_zero] = vec4(c_zero, c_zero, c_zero, c_one);
if(enable_tex[indx_zero])
{
    if(enable_tex_matrix[indx_zero])
        v_texcoord[indx_zero] = tex_matrix[indx_zero] *
                                        a_texcoord0;
    else
        v_texcoord[indx_zero] = a_texcoord0;
}
v_texcoord[indx_one] = vec4(c_zero, c_zero, c_zero, c_one);
if(enable_tex[indx_one])
{
    if(enable_tex_matrix[indx_one])
        v_texcoord[indx_one] = tex_matrix[indx_one] * a_texcoord1;
```

```
        else
            v_texcoord[indx_one] = a_texcoord1;
    }
    v_ucp_factor = enable_ucp ? dot(p_eye, ucp_eqn) : c_zero;
    v_fog_factor = enable_fog ? compute_fog() : c_one;
    gl_Position = mvp_matrix * a_position;
}
```

9 贴图

现在讲完了顶点着色器，你应该熟悉顶点转换的细节，准备基元渲染。管线的下一个步骤是片段着色器。在 OpenGL ES 2.0 中有很多视觉盛宴。可编程着色器是产生这些效果的基础。这些效果包括贴图、像素光照和阴影。片段着色器的基础应用是使用表面贴图。本章讲述创建、装载和应用贴图的所有细节。

贴图基础

装载贴图和材质转换

贴图过滤的包裹

片段着色器上使用贴图

贴图子图像用法

从颜色缓冲器拷贝贴图数据

贴图扩展

贴图基础

3D 图像渲染最基础的操作是使用表面贴图。贴图允许对几何图形没有的网络增加细节。OpenGL ES 2.0 中有两种贴图：2D 贴图和立方体贴图。贴图典型的应用是使用（被认为是贴图矩阵数据的索引）贴图坐标应用到物体的表面。下面的章节介绍不同的贴图类型和它们怎么装载及使用。

2D 贴图

一个 2D 贴图是最基础和普遍的 OpenGL ES 应用。一个 2D 贴图，像你猜测的一样有两个空间的图像数据阵列。贴图的单个数据元素被认为是 texels（纹理）。一个纹理是描述贴图像素的快捷方式。OpenGL ES 中的贴图图像数据能够使用很多不同的基本格式描述。基本的可用的格式看表 9-1.

表 9-1 贴图基本格式

Base Format	Texel Data Description
GL_RGB	(Red, Green, Blue)
GL_RGBA	(Red, Green, Blue, Alpha)
GL_LUMINANCE	(Luminance)
GL_LUMINANCE_ALPHA	(Luminance, Alpha)
GL_ALPHA	(Alpha)

每个图像上的纹理根据它的基本格式和类型指定。一会，我们描述纹理能够使用的各种数据类型的更多的细节。现在我们首先弄懂 2D 贴图是一个两纬空间的图像数据阵列。当渲染 2D 贴图时，贴图的坐标被作为贴图图像的索引。编写 3D 程序时，每个顶点有一个贴图坐标。2D 的贴图坐标写作(s, t)，有时也写作(u, v)。坐标的匹配看图 9-1。（翻译不了，自己体会吧）

贴图图像上的左边和下边的角被认为是 st 坐标系的(0.0, 0.0)，右边和上边的角被认为是(1.0, 1.0)。超过[0.0, 1.0]的坐标也是允许的。获取超过此范围的贴图行为被贴图 texture wrapping mode 定义。（在贴图过滤和 wrapping 章节描述）

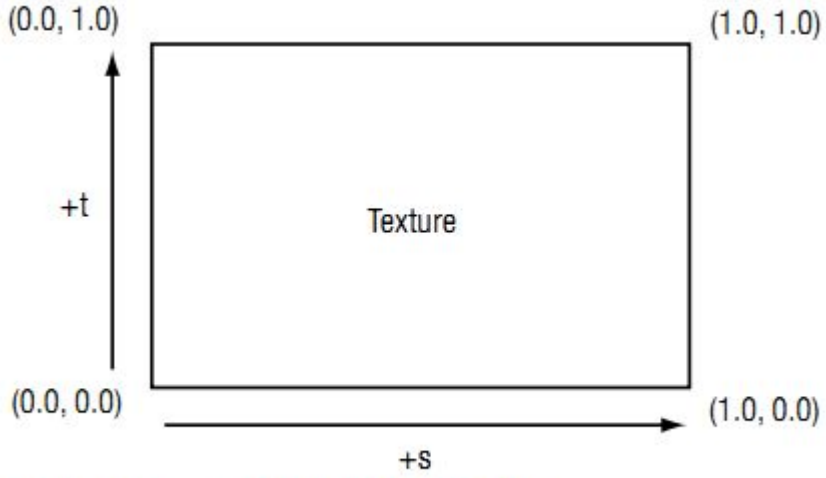


Figure 9-1 2D Texture Coordinates

立方体纹理映射

除了 2D 贴图，OpenGL ES 2.0 也支持立方体纹理贴图。立方体纹理是由六个独立的 2D 贴图组成的一个贴图。每个立方体贴图的一个面代表立方体六个面中的一个。在 3D 渲染时，立方体纹理具有很多优越性。但最基本的用法是环境映射。使用立方体映射去代表环境，这时环境映射到一个渲染的物体上。典型的用法是在场景的中心架设一个计算机，去扑捉六个方向(+X, –X, +Y, –Y, +Z, –Z)图像，存储到一个立方体表面。

纹理贴到立方体的方法是使用 3D 矢量(s,t,r)做贴图坐标匹配立方体。3D 矢量首先被用于选择的立方体的一个面，然后贴图坐标被投射到 2D (s,t)坐标匹配立方体的面。实际的 2D 坐标数学计算超出了我们的范围，但 3D 矢量被使用满足匹配立方体。（不知道怎么翻译才好）。你可以试着想像一个来自立方体内部的 3D 矢量图片。3D 矢量和立方体相交的点就是贴到立方体的纹理。图 9-2 显示了 3D 矢量和立方体面相交的情况。

立方体纹理的每个面以相同的方式独立的指定一个 2D 贴图。每个面必须是正方形的（长宽相等）。3D 矢量被用作贴图坐标时，像 2D 贴图网络一样直接存储每个顶点是不正常的。大多数普遍的吸附到立方体的方法是使用法向量计算立方体坐标贴图。典型的法向量是立方体观察者到反射矢量的方向。这个计算在 13 章 OpenGL ES 2.0 先进的编程中的环境匹配例子中描述。

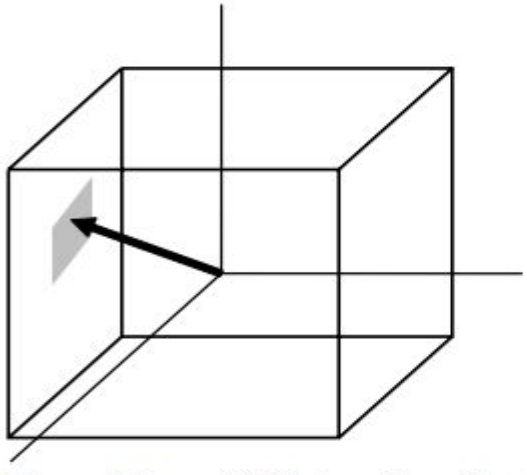


Figure 9-2 3D Texture Coordinate for Cubemap

贴图对象和装载贴图

贴图应用的第一步是创建贴图对象。贴图对象是一个包含着图片数据、过滤模型、包装模型等用于渲染的贴图数据的容器。在 OpenGL ES 中，贴图对象是一个无符号整型的句柄。使用下面的函数产生贴图对象。

void glGenTextures(GLsizei n, GLuint *textures)

n 指定产生的贴图对象的数量

textures 一个存储着贴图对象 Id 值的无符号矩阵

贴图对象不使用时需要删除，例如在应用关闭，游戏里改变场景，需要使用 glDeleteTextures 函数。

void glDeleteTextures(GLsizei n, GLuint *textures)

n 指定被删除的贴图对象的数量

textures 一个存储着需要删除贴图对象 Id 值的无符号矩阵

贴图对象的 ID 必须是 glGenTextures 产生的，应用必须绑定贴图对象才能运行它。一旦贴图对象被绑定，随后的操作像 glTexImage2D 和 glTexParameter 将起作用。绑定使用 glBindTexture 函数。

void glBindTexture(GLenum target, GLuint texture)

target 绑定贴图对象到目标 GL_TEXTURE_2D 或 GL_TEXTURE_CUBE_MAP

texture 贴图对象被绑定的句柄

一旦贴图被绑定到一个特殊的贴图目标，它将保持这种绑定直到目标被删除，产生贴图，绑定后下一步是使用贴图装载实际的图像数据。基本的装载贴图函数是 glTexImage2D。

void glTexImage2D(GLenum target, GLint level,internalFormat, GLsizei width,
GLenum height, GLint border,
GLsizei GLenum format, GLenum type,
const void* pixels)

target 指定贴图对象的目标，是 GL_TEXTURE_2D 或着 立方体的一个面(例如，GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 等等.)

level 指定装载的水平。基本的水平是 0，对连续的多级纹理水平不断增加

internalFormat 纹理存储的内部格式，可以是：
GL_RGBA
GL_RGB
GL_LUMINANCE_ALPHA
GL_LUMINANCE
GL_ALPHA

Width 图像宽度，单位是像素

height 图像高度，单位是像素

border this parameter is ignored in OpenGL ES, but it was kept for compatibility with the desktop OpenGL interface; should be 0

format the format of the incoming texture data. Note that in OpenGL ES the format and internalFormat arguments must have the same value. The supported formats are the same as the internal formats

type 传入像素数据的类型，能够是：
GL_UNSIGNED_BYTE
GL_UNSIGNED_SHORT_4_4_4_4

GL_UNSIGNED_SHORT_5_5_5_1
GL_UNSIGNED_SHORT_5_6_5

pixels 实际图像的像素数据。数据必须包括像素数目（宽×高），每个像素基于格式和类型定义的合适的位。像素行必须依据函数 glPixelStorei (defined next)使用参数 GL_UNPACK_ALIGNMENT 设定

例 9-1 代码描述了 Simple_Texture2D 贴图的例子，展示了产生贴图对象、绑定它、使用 RGB 格式装载 2×2 的 2D 贴图。

例 9-1 产生贴图对象、绑定它、使用 RGB 格式装载 2×2 的 2D 贴图。

```
// Texture object handle
GLuint textureId;

// 2 x 2 Image, 3 bytes per pixel(R, G, B)
GLubyte pixels[4 * 3] =
{
    255,   0,   0, // Red
      0, 255,   0, // Green
      0,   0, 255, // Blue
    255, 255,   0  // Yellow
};
// Use tightly packed data
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

// Generate a texture object
glGenTextures(1, &textureId);
// Bind the texture object
glBindTexture(GL_TEXTURE_2D, textureId);
// Load the texture
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 2, 2, 0, GL_RGB,
             GL_UNSIGNED_BYTE, pixels);

// Set the filtering mode
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

代码第一步像素矩阵被简单的 2×2 初始化，数据无符号范围是[0, 255]的 RGB 数据组成。数据被从着色器的 8 位无符号的贴图组成中取出，值范围[0, 255]匹配浮点范围[0.0, 1.0]。一般情况下，应用不这样简单的创建贴图数据，而是从图像文件里装载数据。例子仅仅是为了演示 API 的使用。

注意调用 glTexImage2D 应用传入一个参数到 glPixelStorei 去设定解压阵列的对齐模式。当贴图数据被 glTexImage2D 装载时，像素的行被认为是对齐的，对齐方式是 GL_UNPACK_ALIGNMENT。默认情况下值是 4，意味着数据行被设定以 4 位一组对齐。应用设定解压对齐是 1，意味着每行数据以一位边界（或者说，数据是紧密排列。glPixelStorei 的定义如下：

void glPixelStorei(GLenum pname, GLint param)

pname 指定像素存储设定的类型，必须设定为 GL_PACK_ALIGNMENT 或者

GL_UNPACK_ALIGNMENT

param　对封装或不封装的数据指定对齐的整形值

glPixelStorei 的参数 GL_PACK_ALIGNMENT 不会影响贴图图像的装载。对齐的包装被 glReadPixels 使用，这在第 11 章片段操作描述，被 glPixelStorei 设定的的包装和未包装队列是全局状态，不要存储或使用贴图对象关联。

回到程序例子，定义过图像数据后，使用 glGenTextures 产生贴图对象。然后这个东西被 glBindTexture 函数绑定到 GL_TEXTURE_2D。最后使用 glTexImage2D 装载。格式被设定为 GL_RGB，这意味着图像数据由三种成分（RGB）构成。类型被设定为 GL_UNSIGNED_BYTE，这意味着数据的每个通道以 8 位无符号类型存储。装载图像也有其它的选项，包括表 9-1 里描述的其他格式。另外，所有纹理的组成能被包装成 16 位，格式有 GL_UNSIGNED_SHORT_4_4_4_4, GL_UNSIGNED_SHORT_5_5_5_1, 或 GL_UNSIGNED_SHORT_5_6_5。

代码最后使用 glTexParameteri 设定缩小和放大过滤模式为 glTexParameteri。我们没有装载完整的一系列缩小图像，只能选择非多级纹理缩小过滤。另外的选项就是使用 GL_LINEAR 的缩小和放大过滤。贴图过滤和多级纹理在后面解释。

贴图过滤和多级纹理

到目前为止，我们限制我们的贴图应用，只是使用了单一的 2D 图像。这允许我们解释贴图的概念，实际 OpenGL ES 中贴图的应用更丰富。使用单一的贴图纹理将会产生可见的痕迹和性能问题。我们到目前为止描述的贴图坐标被用来产生 2D 索引去匹配贴图。当缩小和放大过滤被设定为 GL_NEAREST，将发生这样的事情：一个单一的纹理被匹配到贴图坐标位置。这被认为是一个点或最近的采样。

最近的采样可能会产生一个显著的痕迹。原因是屏幕上的三角形变得非常小。贴图坐标从一个像素到另一个像素插补时产生了大的跳跃。大的贴图产生小的采样数，产生痕迹走样。OpenGL ES 中解决这一问题的方法是 mipmapping（多级纹理）。多级纹理的方法是建立一系列的图像做贴图链。贴图链使用原始图像作第一个，后面的每个图像都只有前面的一半大。链底的最后一个图像是 1×1 的贴图。一般采用计算四个图像的平均产生中间图像的方法 box filtering（盒过滤）。

在第 9 章，2D 多级纹理的例子中，我们提供一个使用 box filtering 技术产生多级纹理链的例子。产生多级纹理的例子通过 GenMipMap2D 实现。这些代码使用 RGB8 图像做输入使用 box filtering 产生下一级纹理。box filtering 怎样执行具体看源码。例 9-2 显示使用 glTexImage2D 装载多级纹理。

例 9-2 装载 2D 多级纹理链

```
// Load mipmap level 0
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height,
             0, GL_RGB, GL_UNSIGNED_BYTE, pixels);

level = 1;
prevImage = &pixels[0];

while(width > 1 && height > 1)
{
   int newWidth,
       newHeight;
   // Generate the next mipmap level
```

```
    GenMipMap2D(prevImage, &newImage, width, height,
                &newWidth, &newHeight);
    // Load the mipmap level
    glTexImage2D(GL_TEXTURE_2D, level, GL_RGB,
                 newWidth, newHeight, 0, GL_RGB,
                 GL_UNSIGNED_BYTE, newImage);
    // Free the previous image
    free(prevImage);
    // Set the previous image for the next iteration
    prevImage = newImage;
    level++;
    // Half the width and height
    width = newWidth;
    height = newHeight;
}
free(newImage);
```

当一个多级纹理链装载，我们可以设定过滤模式，为了达到屏幕像素和贴图图片像素更好的配合比例，减少锯齿痕迹。因为多级纹理贴图的成功过滤，当在更远处观察时，我们向贴图链后移动，锯齿减少，实现高质量的图像。

使用贴图时，有两种类型的过滤：缩小和放大。屏幕上物体表面积小于贴图尺寸时使用缩小过滤。屏幕上物体表面积大于贴图尺寸时使用放大过滤。那种类型的过滤被使用由硬件决定。但 API 提供在任何情况下的过滤类型。放大处理是不相关的，因为我们总是使用最大可用的级别。对于缩小，有各种 的采样模式可以使用。那种模式被使用选择基于你需要实现的可视质量和你想要实现多大性能的纹理过滤。

过滤模式使用 glTexParameter[i|f][v]指定。

```
void   glTexParameteri(GLenum target, GLenum pname, GLint param)
void   glTexParameteriv(GLenum target, GLenum pname, const GLint *params)
void   glTexParameterf(GLenum target, GLenum pname, GLfloat param)
void   glTexParameterfv(GLenum target, GLenum pname, const GLfloat *params)
target  绑定贴图对象的目标 GL_TEXTURE_2D 或 GL_TEXTURE_CUBE_MAP
pname   设定的参数，可能值是：
        GL_TEXTURE_MAG_FILTER
        GL_TEXTURE_MIN_FILTER
        GL_TEXTURE_WRAP_S
        GL_TEXTURE_WRAP_T
params  设定贴图参数的值(对 v 入口点的矩阵值)，
        如果 pname 是 GL_TEXTURE_MAG_FILTER, 那么 param 是:
        GL_NEAREST
        GL_LINEAR
        如果 pname 是 GL_TEXTURE_MIN_FILTER, 那么 param 是:
        GL_NEAREST
        GL_LINEAR
        GL_NEAREST_MIPMAP_NEAREST
        GL_NEAREST_MIPMAP_LINEAR
```

GL_LINEAR_MIPMAP_NEAREST
GL_LINEAR_MIPMAP_LINEAR
如果 pname 是 GL_TEXTURE_WRAP_S 或 GL_TEXTURE_WRAP_R, 那么 param 是:
GL_REPEAT
GL_CLAMP_TO_EDGE
GL_MIRRORED_REPEAT

放大过滤是 GL_NEAREST 或 GL_LINEAR。在 GL_NEAREST 放大过滤模式，贴图最近的采样点将使用做贴图坐标。在 GL_LINEAR 模式，双线性（四个点平均）采样点作为贴图坐标。

缩小过滤在下面的情况下被设定：

GL_NEARESTA　　贴图最近的采样点将使用做贴图坐标.
GL_LINEARA　　双线性（四个点平均）采样点作为贴图坐标.
GL_NEAREST_MIPMAP_NEARESTA　　采样点采用最接近 mip 水平.
GL_NEAREST_MIPMAP_LINEARWill　　两个最 mip 的点插补.
GL_LINEAR_MIPMAP_NEARESTWill　　最 mip 水平双线性
GL_LINEAR_MIPMAP_LINEARWill　　两个最 mip 的点双线性，最后的模式被认为是三线性，实现最好的质量。

注意：GL_NEAREST 和 GL_LINEAR 是不要求完整的多级纹理。其它模式的过滤要求完整的贴图链。

2D 多级纹理的例子如图 9-3，显示了使用 GL_NEAREST 和 GL_LINEAR_MIPMAP_LINEAR 绘制多边形。

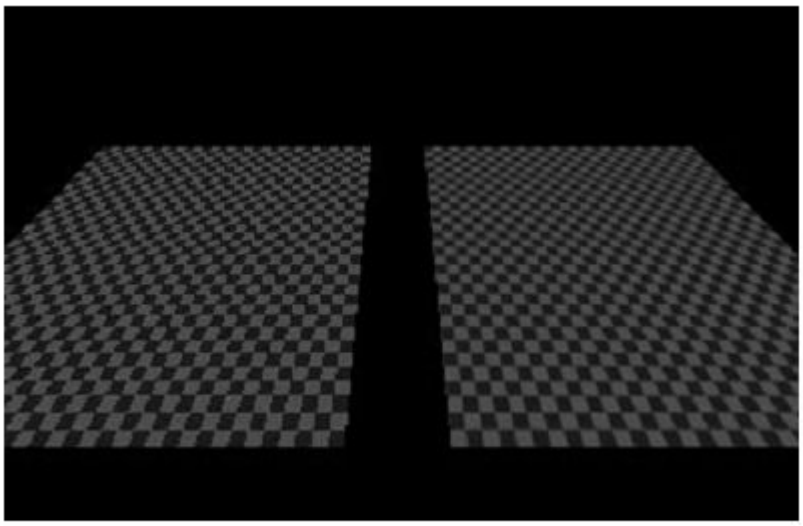


Figure 9-3　MipMap2D: Nearest Versus Trilinear Filtering

一些操作暗示你要选择的贴图过滤模式，如果发生缩小，使用多级纹理过滤对大多数的硬件都是最好的。原因是你不使用多级纹理只有很少的贴图缓冲区可用，获取发生在离散的位置而自始至终都只有一个图。无论怎样，你使用更高质量的模式，硬件执行的代价就更高。例如大多数的硬件双线性对三线性执行代价更低。你应该选择那种模式对你的执行不能产生过大的消极影响。如果更好的贴图过滤不是你的障碍，你可能自由的使用更好的过滤模式。有时候需要调整你的程序和硬件。

自动多级纹理产生

先前章节 2D 多级纹理的例子中，应用程序创建一个级别到 0 的多级纹理链，通过每个图片使用过滤和对宽和高减半产生多级纹理链。这是一种产生多级纹理的方法，但 OpenGL ES

2.0 也支持使用 glGenerateMipmap 自动产生多级纹理的机制。

void glGenerateMipmap(GLenum target)

target 产生多级纹理的目标 可以是 GL_TEXTURE_2D 或 GL_TEXTURE_CUBE_MAP

我们对绑定的贴图对象调用 glGenerateMipmap。这个函数会产生从原始图像到级别为 0 的全部的多级纹理链。对 2D 纹理这意味着级别为 0 的贴图内容成功过滤，可以使用每个子级别的纹理。对立方体贴图，立方体的每个面将产生级别到 0 的多级纹理链。注意使用这种行为，你必须为每个面指定 0 级别，每个面必须有匹配的内部宽和高格式。另一个需要注意的是 OpenGL ES 2.0 没有授权任何产生多级纹理的过滤算法（虽然提到盒过滤）。如果你要求特定的过滤模式，你需要自己产生多级纹理。

当你使用帧缓冲区渲染贴图时，自动多级纹理产生变得特别重要。渲染一个贴图时，我们不想必须读回贴图到 CPU 去产生多级纹理，代替的，glGenerateMipmap 被使用绘图硬件不回读数据到 CPU 就能够产生多级纹理。我们在第 12 章帧缓冲区对象详细介绍，那时这个问题将清晰。

纹理坐标包装

当纹理坐标超过了范围[0.0, 1.0]时，纹理包装被使用。纹理包装模式使用 glTexParameter[i|f][v]设定，纹理包装模式能被独立的设定为 s 坐标和 t 坐标。GL_TEXTURE_WRAP_S 定义 s 坐标超出范围[0.0, 1.0]的情况，GL_TEXTURE_WRAP_T 设定 t 坐标。有三种包装模式供选择，看表 9-2

表 9-2 纹理坐标包装

Texture Wrap Mode	Description
GL_REPEAT	重复纹理
GL_CLAMP_TO_EDGE	采样纹理边缘
GL_MIRRORED_REPEAT	重复纹理和镜像

注意，纹理包装模式对过滤行为有影响。例如纹理坐标是边缘采用时，双线性过滤将扫描纹理的边缘。这时包装模式将决定哪个纹理是纹理边缘的外面而应用于过滤算法。如果你不想要任何格式的重复，应该使用 GL_CLAMP_TO_EDGE。

第九章纹理包装有一个例子使用三种包装模式绘制一个正方形。例 9-4 正方形贴图是贴图坐标范围为[-1.0, 2.0]的西洋跳棋。

Figure 9-4 GL_REPEAT, GL_CLAMP_TO_EDGE, and GL_MIRRORED_REPEAT Modes

三个正方形 使用贴图包装模式代码如下;

```
// Draw left quad with repeat wrap mode
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glUniform1f(userData->offsetLoc, -0.7f);
```

```
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, indices);
// Draw middle quad with clamp to edge wrap mode
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glUniform1f(userData->offsetLoc, 0.0f);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, indices);
// Draw right quad with mirrored repeat
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,
                GL_MIRRORED_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,
                GL_MIRRORED_REPEAT);
glUniform1f(userData->offsetLoc, 0.7f);
glDrawElements GL_TRIANGLES, 6, GL_UNSIGNED_SHORT, indices);
```

图 9-4，左边的使用 GL_REPEAT 模式渲染，超出[0, 1]范围的重复图像的方块样式。中心的正方形使用 GL_CLAMP_TO_EDGE 模型，像你看到的，超出[0, 1]范围的贴图坐标采用图像的边缘。右边的使用 GL_MIRRORED_REPEAT 模式渲染，超出[0, 1]范围的镜像和重复图像的方块样式。

片段着色器里使用贴图

现在我们已经讲完了基础的贴图设置，让我们看一些着色器源码例子。例 9-3 简单的 2D 贴图采样显示了一对顶点、片段着色器 2D 贴图在着色器里的使用。

例 9-3 顶点、片段着色器执行 2D 贴图

```
GLbyte vShaderStr[] =
   "attribute vec4 a_position;   \n"
   "attribute vec2 a_texCoord;   \n"
   "varying vec2 v_texCoord;     \n"
   "void main()                  \n"
   "{                            \n"
   "   gl_Position = a_position; \n"
   "   v_texCoord = a_texCoord;  \n"
   "}                            \n";

GLbyte fShaderStr[] =
   "precision mediump float;                            \n"
   "varying vec2 v_texCoord;                            \n"
   "uniform sampler2D s_texture;                        \n"
   "void main()                                         \n"
   "{                                                   \n"
   "  gl_FragColor = texture2D(s_texture, v_texCoord); \n"
   "}                                                   \n";
```

顶点使用两个贴图坐标作为顶点属性通过变量传递输入到片段着色器。片段着色器使用这些变量作为贴图坐标进行贴图。片段着色器宣布一个称为 sampler2D 类型的特殊的 uniform 叫做 s_texture。采样器是特殊的用来匹配贴图的 uniform 变量。采样器 uniform 将装载指定的已经绑定过的贴图单元。例如从 GL_TEXTURE0 单位指定一个值为 0 的采样器，从 GL_TEXTURE1 指定值为 1，等等。指定贴图绑定到贴图单元的方法是 glActiveTexture 函数。

```
void    glActiveTexture(GLenum texture)
```
texturethe 激活的贴图单元, GL_TEXTURE0, GL_TEXTURE1,…,GL_TEXTURE31

glActiveTexture 函数设定当前的贴图单元，以至于后面的函数 glBindTexture 绑定贴图到当前的单元。一个应用里可用的贴图数目能使用 GL_MAX_TEXTURE_IMAGE_UNITS 为参数调用 glGetIntegeriv 查询。

下面的简单的 2D 贴图例子代码显示采样器和贴图怎样绑定到贴图单元。

```
// Get the sampler locations
userData->samplerLoc = glGetUniformLocation(
                                userData->programObject,
                                "s_texture");
//
// Bind the texture
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, userData->textureId);
// Set the sampler texture unit to 0
glUniform1i(userData->samplerLoc, 0);
```

最后我们来装载贴图，贴图被绑定到贴图单位 0，采样器被设定使用贴图单元 0.回到片段着色器 2D 贴图采样器的例子，我们看使用内建函数 texture2D 去获取贴图匹配的着色器源码。texture2D 函数如下。

```
vec4   texture2D(sampler2D sampler, vec2 coord[,
                    float bias])
```

sampler	一个指定贴图格式绑定到贴图单元的采样器
coord	使用去获取贴图匹配的 2D 贴图坐标
bias	可选的参数，为获取贴图的多级纹理偏置。允许着色器解释计算 LOD 值的偏置，被用于多级纹理选择。

一个 texture2D 返回一个代表从贴图取得的映射的 vec4。贴图数据映射颜色的方法依靠贴图的格式。表 9-3 显示贴图格式匹配 vec4 颜色的方法。

表 9-3 贴图格式和颜色的匹配

Base Format	Texel Data Description
GL_RGB	(R, G, B, 1.0)
GL_RGBA	(R, G, B, A)
GL_LUMINANCE	(L, L, L, 1.0)
GL_LUMINANCE_ALPHA	(L, L, L, A)
GL_ALPHA	(0.0, 0.0, 0.0, A)

在 Simple_Texture2D 例子中，贴图以 GL_RGB 格式被装载，vec4 里是(R, G, B, 1.0)。

使用立方体贴图的例子

使用立方体贴图和使用 2D 贴图一样简单。简单立方体贴图匹配显示使用立方体贴图绘制球体。立方体有六个 1×1 的面，每个有不同的颜色。代码 9-4 是怎样装载立方体贴图。

例 9-4 装载立方体贴图

```
GLuint CreateSimpleTextureCubemap()
{
   GLuint textureId;
   // Six 1 x 1 RGB faces
   GLubyte cubePixels[6][3] =
```

```
    {
       // Face 0 - Red
       255, 0, 0,
       // Face 1 - Green,
       0, 255, 0,
       // Face 3 - Blue
       0, 0, 255,
       // Face 4 - Yellow
       255, 255, 0,
       // Face 5 - Purple
       255, 0, 255,
       // Face 6 - White
       255, 255, 255
    };
      // Generate a texture object
  glGenTextures(1, &textureId);
  // Bind the texture object
  glBindTexture(GL_TEXTURE_CUBE_MAP, textureId);

  // Load the cube face - Positive X
  glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X, 0, GL_RGB, 1, 1,
               0, GL_RGB, GL_UNSIGNED_BYTE, &cubePixels[0]);
  // Load the cube face - Negative X
  glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 0, GL_RGB, 1, 1,
               0, GL_RGB, GL_UNSIGNED_BYTE, &cubePixels[1]);
  // Load the cube face - Positive Y
  glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Y, 0, GL_RGB, 1, 1,
               0, GL_RGB, GL_UNSIGNED_BYTE, &cubePixels[2]);
  // Load the cube face - Negative Y
  glTexImage2D(GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, 0, GL_RGB, 1, 1,
               0, GL_RGB, GL_UNSIGNED_BYTE, &cubePixels[3]);
  // Load the cube face - Positive Z
  glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_Z, 0, GL_RGB, 1, 1,
               0, GL_RGB, GL_UNSIGNED_BYTE, &cubePixels[4]);
   // Set the filtering mode
   glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MIN_FILTER,
                   GL_NEAREST);
   glTexParameteri(GL_TEXTURE_CUBE_MAP, GL_TEXTURE_MAG_FILTER,
                   GL_NEAREST);
   return textureId;
}
```

这个代码使用 glTexImage2D 函数装载 1×1 的 RGB 像素数据到立方体的每个独立的面。渲染这个球体的代码在例子 9-5 提供。

例 9-5 顶点着色器和片段着色器对立方体贴图的搭配

```
GLbyte vShaderStr[] =
   "attribute vec4 a_position;   \n"
   "attribute vec3 a_normal;     \n"
   "varying vec3 v_normal;       \n"
   "void main()                  \n"
   "{                            \n"
   "   gl_Position = a_position; \n"
   "   v_normal = a_normal;      \n"
   "}                            \n";

GLbyte fShaderStr[] =
   "precision mediump float;                          \n"
   "varying vec3 v_normal;                            \n"
   "uniform samplerCube s_texture;                    \n"
   "void main()                                       \n"
   "{                                                 \n"
   "  gl_FragColor = textureCube(s_texture, v_normal); \n"
   "}                                                 \n";
```

顶点着色器使用一个位置和法线作为顶点的属性。一个球体每个顶点的法线被存储用作贴图坐标。法线以变量的形式输入片段着色器。片段着色器使用内建函数 textureCube 去进行贴图，法线作为贴图坐标。函数行为如下：

```
vec4   textureCube(samplerCube sampler, vec3 coord[,float bias])
```

sampler　　绑定到贴图单元的采样器，指定获取的贴图

coord　　被用于立方体贴图的 3D 贴图坐标

bias　　用于获取贴图提供多级纹理偏置的可选参数，允许着色器解释计算 LOD 值的偏置，被用于多级纹理选择。

立方体贴图行为和 2D 贴图非常类似。区别是贴图坐标是三个成员而不是两个，贴图类型是立方体采样。使用绑定立方体贴图和装载采样器的方法和 Simple_Texture2D 例子是相似的。

压缩贴图

至今为止，我们使用 glTexImage2D 处理贴图仅仅装载未压缩的图像。OpenGL ES 2.0 也支持装载压锁的贴图图像数据。支持压缩的图像有这样几个值得原因。首先很明显压缩图像减少设备内存的使用。第二，压缩图像节省带宽的使用。最后压缩图像减少存储应用图像数据的。

OpenGL ES 2.0 规范没有定义任何压缩图像格式，OpenGL ES 2.0 定义了压缩图像能被装载的机制。OpenGL ES 2.0 的商家需要提供扩展支持压缩图像数据类型。Ericsson Texture Compression (ETC)是一个被批准的可选的扩展，已经被一些商家支持。像 AMD、ARM、Imagination Technologies、NVIDIA 也提供他们自己硬件支持的扩展。

glCompressedTexImage2D 是为 2D 贴图和立方体贴图装载压缩图像数据的函数。

```
void   glCompressedTexImage2D(GLenum target, GLint level,
                  GLenum  internalformat, GLsizei width, GLenum height,
                  GLint border, GLsizei imageSize, const void *data)
```

target　　指定贴图目标，应该是 GL_TEXTURE_2D 或 GL_TEXTURE_CUBE_MAP_* 面中的一个目标

level　　指定多级纹理的那一级被装载，基本的等级是 0，对随后的多级纹理，依次增加

internalFormat　贴图存储的内部格式，这是使用的压缩贴图的格式。非压缩图像格式被 OpenGL ES 2.0 指定，这里的格式必须是来自扩展格式。

width　　贴图以像素为单位的宽度

height　　贴图以像素为单位的高度

border　　这个参数在 OpenGL ES 中被忽略，保留它仅仅是为了和桌面版本兼容，这里必须设置为 0

imageSize　图像以 bytes 为单位的尺寸

data　　贴图的压缩后的数据，数据数量必须和 imageSize 数据尺寸一致

一旦压缩贴图被装载，它能够像非压缩贴图一样被正确的使用。注意如果你使用了 OpenGL ES 2.0 不支持的压缩格式，会产生一个 GL_INVALID_ENUM 错误。为你使用的压缩扩展格式做字符串检查是非常重要的，如果不这样做，你最好使用非压缩图像格式。

除了检查扩展字符串，还有另一个方法你能使用决定编译器支持的压缩格式。你能使用 glGetIntegerv 查询 GL_COMPRESSED_TEXTURE_FORMATS，它返回一个 Glenum 值数组，每一个 Glenum 值都是一个编译工具支持的压缩贴图格式。

贴图子图像规格

使用 glTexImage2D 装载一个贴图图像后，可能需要更新部分区域的图像。如果你仅仅需要更新一个图像的子区域，使用 glTexSubImage2D 函数。

```
void   glTexSubImage2D(GLenum target, GLint level,
                       GLint xoffset, GLint yoffset,
                       GLsizei width, GLsizei height,
                       GLenum format, GLenum type,
                       const void* pixels)
```

target　指定贴图目标，可以是 GL_TEXTURE_2D 或者是立方体贴图的一个面（例如，GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 等等.）

level　指定哪一级水平被更新

xoffset　像素开始更新的 x 索引起始地址

yoffset　像素开始更新的 y 索引起始地址

width　更新图像的子区域宽度

height　更新图像的子区域高度

format　输入的贴图数据格式，可能是：

GL_RGBA

GL_RGB

GL_LUMINANCE_ALPHA

GL_LUMINANCE

GL_ALPHA

type　输入的像素数据类型，可能是：

GL_UNSIGNED_BYTE

GL_UNSIGNED_SHORT_4_4_4_4

GL_UNSIGNED_SHORT_5_5_5_1

GL_UNSIGNED_SHORT_5_6_5

pixels　贴图子区域的包含的用于更新的像素数据

这个函数将更新(xoffset, yoffset)到(xoffset + width -1,yoffset + height -1)范围内的纹理。注意使用这个函数，贴图必须已经被指定。子图像区域必须在先前指定的贴图图像范围内。纹理的数组的格式必须是函数glPixelStorei用GL_UNPACK_ALIGNMENT参数指定的。

更新压缩 2D 贴图图像子区域的函数是 glCompressedTexSubImage2D。这个函数定义和 glTexImage2D 相似。

```
void   glCompressedTexSubImage2D(GLenum target, GLint level,
                                 GLint xoffset, GLint yoffset,
                                 GLsizei width, GLsizei height,
                                 GLenum format,GLenum imageSize
                                 const void* pixels)
```

target 指定贴图目标，可以是 GL_TEXTURE_2D 或者是立方体贴图的一个面(例如, GL_TEXTURE_CUBE_MAP_POSITIVE_X,GL_TEXTURE_CUBE_MAP_NEGATIVE_X,等等.)

level 指定哪一级水平被更新

xoffset 像素开始更新的 x 索引起始地址

yoffset 像素开始更新的 y 索引起始地址

width 更新图像的子区域宽度

height 更新图像的子区域高度

format 使用的压缩贴图格式. 非压缩贴图格式被 OpenGL ES 2.0 核心定义,这里的格式必须来自扩展，匹配的格式是前面图像已指定的格式。

pixels 实际的图像子区域的像素数据

从颜色缓冲区拷贝贴图数据

OpenGL ES 2.0 支持的另一个贴图特性是能从颜色缓冲区拷贝数据做贴图。如果你想使用渲染的结果作为贴图的图像就使用它。注意帧缓冲区（12 章）提供一个快速的方法处理渲染到贴图，这样做比拷贝图像数据更快。如果不是这种操作，从颜色缓冲区拷贝贴图数据也是一个有用的特性。

OpenGL ES 2.0 仅仅支持双缓冲区 EGL 显示表面。这意味着所有 OpenGL ES 2.0 应用都在前和后两个缓冲区绘制显示。当前缓冲区是前还是后被 eglSwapBuffers 最近一次的调用决定。当你从显示的 EGL 窗口的颜色缓冲区拷贝数据时，你总是从后缓冲区拷贝内容。如果你正在渲染一个 EGL pbuffer，拷贝将在 pbuffer 发生。如果你正在渲染一个缓冲区目标，拷贝将在帧缓冲区代替颜色缓冲区。

从颜色缓冲区拷贝数据到贴图函数是 glCopyTexImage2D 和 glCopyTexSubImage2D。

```
void   glCopyTexImage2D(GLenum target, GLint level,
                        GLenum internalFormat, GLint x,
                        GLint y, GLsizei width,
                        GLsizei height, GLint border )
```

target 指定贴图目标，可以是 GL_TEXTURE_2D 或者是立方体贴图的一个面 (例如, GL_TEXTURE_CUBE_MAP_POSITIVE_X,GL_TEXTURE_CUBE_MAP_NEGATIVE_X,等等.)

level 指定哪一级水平被装载

internalFormat 图像的内部格式，可以是：

- GL_RGBA
- GL_RGB
- GL_LUMINANCE_ALPHA
- GL_LUMINANCE
- GL_ALPHA

x　　从帧缓冲区读出的正方形的下面和左边的 x 窗口坐标
y　　从帧缓冲区读出的正方形的下面和左边的 y 窗口坐标
width　读出的区域以像素为单位的宽度
height　读出的区域以像素为单位的高度
border　在 OpenGL ES 2.0 中不支持 borders,这个参数必须设定为 0

调用这个函数，贴图图像将装载颜色缓冲区范围(x,y) 到(x + width -1,y + height-1)数据到图像纹理。贴图图像的高和宽是从颜色缓冲区区域拷贝的尺寸。你能使用这去填充整个贴图内容。

你能仅仅使用 glCopyTexSubImage2D 更新一个子区域。

```
void    glCopyTexSubImage2D(GLenum target, GLint level,
                            GLint level, GLint xoffset,
                            GLint yoffset, GLint x, GLint y,
                            GLsizei width, GLsizei height)
```

target　指定贴图目标，可以是 GL_TEXTURE_2D 或者是立方体贴图的一个面(例如,
　　GL_TEXTURE_CUBE_MAP_POSITIVE_X,GL_TEXTURE_CUBE_MAP_NEGATIVE_X,等等.)
level　指定哪一级水平被装载
xoffset　像素开始更新的 x 索引起始地址
yoffset　像素开始更新的 y 索引起始地址
x　　从帧缓冲区读出的正方形的下面和左边的 x 窗口坐标
y　　从帧缓冲区读出的正方形的下面和左边的 y 窗口坐标
width　读出的区域以像素为单位的宽度
height　读出的区域以像素为单位的高度

这个函数将使用颜色缓冲区(x, y)到(x + width-1,y + height-1)中的像素，更新(xoffset,yoffset)到(xoffset + width-1,yoffset + height-1)子区域的图像。

一个需要注意的事情是使用 glCopyTexImage2D 和 glCopyTexSubImage2D 的图像格式不能比颜色缓冲区组成复杂。换句话说当从颜色缓冲区拷贝数据时，可能转变格式为更简单的格式，但却不能相反。表 9-4 显示了拷贝图像时有效的数据组成。从表中我们可以看到拷贝 RGBA 格式的图像能够转变成其他各种格式，但不能拷贝 RGB 数据转变成 RGBA 图像，因为颜色缓冲区里不存在透明度成分。

Table 9-4　Valid Format Conversions for `glCopyTex*Image2D`

	Texture Format				
Color Format	A	L	LA	RGB	RGBA
A	Y	N	N	N	N
L	N	Y	N	N	N
LA	Y	Y	Y	N	N
RGB	N	Y	N	Y	N
RGBA	Y	Y	Y	Y	Y

可选的扩展

有一些Khronos组织支持的扩展提供附加的贴图行为可以在OpenGL ES 2.0核心里使用。这些扩展提供对 3D 贴图、浮点贴图、爱立信压缩贴图和非 2 的指数次方贴图。下面的章节描述 Khronos 组织批准的扩展。

3D 贴图

除了 2D 贴图和立方体贴图，有一些 OpenGL ES 2.0 的 3D 贴图扩展 GL_OES_texture_3D。这个扩展方法装载和渲染 3D 贴图（或体积贴图）。3D 贴图可以被认为是 2D 贴图的切片矩阵。一个 3D 贴图有三个坐标(s,t,r)象立方体贴图。对 3D 贴图 r 坐标用来选择 3D 坐标的那个切片被用来采样，(s,t)坐标被用于获取切片的 2D 贴图。每个 3D 贴图的多级纹理包含着一部分的贴图的切片。

装载 3D 贴图的命令是 glTexImage3DOES，它和 glTexImage2D 非常相似。

```
void   glTexImage3DOES(GLenum target, GLint level,
                        GLenum  internalFormat, GLsizei width,
                       GLsizei height, GLsizei depth,
                       GLint border, GLenum format,
                       GLenum type, const void*  pixels)
```

target 指定贴图目标， 应该是 GL_TEXTURE_3D_OES

level 指定哪一级水平被更新，基本的等级是 0，对随后的多级纹理，依次增加

internalFormat 贴图存储的内部格式:

GL_RGBA
GL_RGB
GL_LUMINANCE_ALPHA
GL_LUMINANCE
GL_ALPHA

width 贴图以像素为单位的宽度

height 贴图以像素为单位的高度

depth 3D 贴图切片的数

border 这个参数在 OpenGL ES 中被忽略，保留它仅仅是为了和桌面版本兼容，这里必须设置为 0

format 输入贴图数据的格式，OpenGL ES 2.0 的格式和内部格式参数必须相同，支持的格式和内部格式相同。

type 输入的像素数据类型，可能是:

GL_UNSIGNED_BYTE
GL_UNSIGNED_SHORT_4_4_4_4
GL_UNSIGNED_SHORT_5_5_5_1
GL_UNSIGNED_SHORT_5_6_5

pixels 贴图的实际像素数据，数据必须有（宽×高×深度）数目的像素，像素格式基于指定的格式和类型。像素数据应该被以 2D 贴图切片顺序存储。

一旦 3D 贴图使用 glTexImage3DOES 装载，贴图能在着色器里使用 3D 贴图的内在功能。在使用这之前，着色器必须使用#extension 机制使能 3D 贴图扩展。

```
#extension GL_OES_texture_3D : enable
```

在片段着色器内，扩展被使能后，着色器能使用 3D 贴图的内建函数，使用的函数是：

```
vec4   texture3D(sampler3D sampler, vec3 coord[, float bias])
```

sampler 绑定贴图单元，指定贴图获取的采样器

coord 使用去获取贴图匹配的 3D 贴图坐标

bias 用于获取贴图提供多级纹理偏置的可选参数，允许着色器解释计算 LOD 值的偏置，被用于多级纹理选择。

注意到 r 坐标是浮点值，依靠设定的过滤模式，贴图可能获取跨越两个系列切片。3D

贴图扩展也支持 GL_TEXTURE_WRAP_R_OES，这个参数被用于设定 r 坐标包装模式（象 2D 贴图的 s 或 t 坐标包装模式）。另外这个扩展也支持装载压缩 3D 贴图数据。压缩贴图数据能使用 glCompressedTexImage3DOES 装载。象 2D 压缩贴图，这个扩展也没有指定 3D 贴图格式。

```
void    glCompressedTexImage3DOES(GLenum target, GLint level,
                                  GLenum internalformat,
                                  GLsizei width, GLsizei height,
                                  GLsizei depth, GLint border,
                                  GLsizei imageSize, const void *data)
```

target 指定贴图目标，必须设定为 GL_TEXTURE_3D

level 指定哪一级水平被装载，基本的等级是 0，对随后的多级纹理，依次增加

internalFormat 贴图存储的内部格式，这是压缩贴图格式，非压缩图象的格式被 OES_texture_3D 定义，这里使用的压缩格式必须来自另一个扩展。

width 贴图以像素为单位的宽度

height 贴图以像素为单位的高度

depth 贴图以像素为单位的深度

border 这个参数在 OpenGL ES 中被忽略，保留它仅仅是为了和桌面版本兼容，这里必须设置为 0

imageSize 贴图图像以 byte 为单位的尺寸

data 贴图的压缩后的数据，数据数量必须和 imageSize 数据尺寸一致

另外，象 2D 贴图一样，可能更新 3D 贴图的一个子区域，使用 glTexSubImage3DOES.函数

```
void    glTexSubImage3DOES(GLenum target, GLint level,
                           GLint xoffset, GLint yoffset,
                           GLint zoffset, GLsizei width,
                           GLsizei height, GLsizei depth,
                           GLenum format, GLenum type,
                           const void* pixels)
```

target 指定贴图目标，可以是 GL_TEXTURE_2D 或者是立方体贴图的一个面(例如，GL_TEXTURE_CUBE_MAP_POSITIVE_X，GL_TEXTURE_CUBE_MAP_NEGATIVE_X，等等.)

level 指定哪一级水平被更新

xoffset 像素开始更新的 x 索引起始地址

yoffset 像素开始更新的 y 索引起始地址

zoffset 像素开始更新的 z 索引起始地址

width 更新图像的子区域宽度

height 更新图像的子区域高度

depth 更新图像的子区域深度

format 输入的像素数据类型，可能是:

- GL_RGBA
- GL_RGB
- GL_LUMINANCE_ALPHA
- GL_LUMINANCE
- GL_ALPHA

type 输入的像素数据类型，可能是:

- GL_UNSIGNED_BYTE

GL_UNSIGNED_SHORT_4_4_4_4
GL_UNSIGNED_SHORT_5_5_5_1
GL_UNSIGNED_SHORT_5_6_5

pixels 贴图子区域的包含的用于更新的像素数据

glTexSubImage3DOES 函数象 glTexSubImage2D 一样，前面已经讲过了。仅仅的不同是子区域包括指定子区域的 zoffset 和深度，指定更新深度内的切片。对压缩 3D 贴图，可能使用 glCompressedTexSubImage3DOES 更新贴图的子区域。

```
void    glCompressedTexSubImage3DOES(GLenum target,
                                     GLint level, GLint xoffset,
                                     GLint yoffset, GLint zoffset,
                                     GLsizei width, GLsizei height,
                                     GLsizei depth, GLenum format,
                                     GLenum imageSize,
                                     const void* data)
```

target 指定贴图目标，可以是 GL_TEXTURE_2D 或者是立方体贴图的一个面(例如,
GL_TEXTURE_CUBE_MAP_POSITIVE_X,
GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 等等.)

level 指定哪一级水平被更新

xoffset 像素纹理开始更新的 x 索引起始地址

yoffset 像素纹理开始更新的 y 索引起始地址

zoffset 像素纹理开始更新的 z 索引起始地址

width 更新图像的子区域宽度

height 更新图像的子区域高度

depth 更新图像的子区域深度

format 使用的压缩贴图格式. 非压缩贴图格式被 OpenGL ES 2.0 核心定义,这里的格式必须来自扩展，匹配的格式是前面图像已指定的格式。

data 实际的图像子区域的像素数据

最后，也可以使用 glCopyTexSubImage3DOES 从颜色缓冲区拷贝内容到先前指定的 3D 贴图切片（或子区域切片）。

```
void    glCopyTexSubImage3DOES(GLenum target, GLint level,
                               GLint level, GLint xoffset,
                               GLint yoffset, GLint zoffset,
                               GLint x, GLint y,
                               GLsizei width, GLsizei height)
```

target 指定贴图目标，可以是 GL_TEXTURE_2D 或者是立方体贴图的一个面(例如,
GL_TEXTURE_CUBE_MAP_POSITIVE_X,
GL_TEXTURE_CUBE_MAP_NEGATIVE_X, 等等.)

level 指定哪一级水平被装载

xoffset 像素开始更新的 x 索引起始地址

yoffset 像素开始更新的 y 索引起始地址

zoffset 像素开始更新的 z 索引起始地址

x 从帧缓冲区读出的正方形的下面和左边的 x 窗口坐标

y 从帧缓冲区读出的正方形的下面和左边的 x 窗口坐标

width 读出的区域以像素为单位的宽度

height　读出的区域以像素为单位的宽度

这个函数和 glCopyTexSubImage2D 有相同的参数，不同是 zoffset 参数。这个参数选择拷贝颜色缓冲区内容到的是那个切片。函数执行结果是颜色缓冲区内容和 2D 贴图切片内内容相同。

爱立信压缩贴图

GL_OES_compressed_ETC1_RGB8_texture 是被批准的 OpenGL ES 2.0 压缩贴图扩展。这个扩展使用的压缩技术是 ETC 格式。ETC 格式是基于块压缩的 RGB 贴图。ETC 格式存储 4×4 的纹理块，每个 64 位大小。这意味着原始图像是 24 位的 RGB 数据，使用 ETC 压缩率是 6：1.这个格式布局细节超出了本书的范围。然而有一些自由的工具想 AMD 的 Compressonator 能使用 ETC 压缩图像。一旦压缩图像被产生，它可以使用 glCompressedTexImage2D 函数装载。

浮点贴图

在 OpenGL ES 2.0 中，没有方法使用 16 位或 32 位浮点精度存储图像。扩展 GL_OES_texture_half_float 和 GL_OES_texture_float 支持 16 位和 32 位浮点贴图。贴图能够使用 GL_HALF_FLOAT_OES 和 GL_FLOAT_OES 装载。扩展 GL_OES_texture_half_float_linear 和 GL_OES_texture_float_linear 表明浮点贴图使用最近的采样（双线性、三线性）过滤图像。16 位浮点值格式有 1 位是信号位，5 位指数位，10 位小数位。详细的描述看附录 A。

非 2 指数倍数的贴图

OpenGL ES 2.0 中，贴图可以是 non-power-of-two (npot) 维数。换句话说，宽和高可以不需要是 2 的维数倍。如果贴图维数不是 2 的指数次方，OpenGL ES 2.0 在包装模式上有一个限制。这就是说，包装模式不是 2 的指数倍数，包装模式仅仅能使用 GL_CLAMP_TO_EDGE 和缩小过滤仅仅能使用 GL_NEAREST 或 GL_LINEAR(或者说非多级纹理)。GL_OES_texture_npot 扩展突破这个限制，允许 GL_REPEAT 和 GL_MIRRORED_REPEAT 模式，也允许 npot 贴图是多级纹理使用全缩小过滤设定。

片段着色器

第 9 章贴图我们介绍了在片段着色器里创建和应用基本的贴图。本章我们讲述更多片段着色器和它的使用。特别的我们专注于使用片段着色器执行固定功能行为的技术。我们在这章要讲述的内容包括：

固定功能行为片段着色器

片段着色器整体介绍

混合贴图

雾

透明度测试

使用剪切平面

看图 10-1 的管道图表，我们已经讲述了管道的顶点着色器、基元装配、光栅化阶段，我们谈到在片段着色器里使用贴图。现在我们开始关心管道的片段着色器和写片段着色器的剩余的细节。

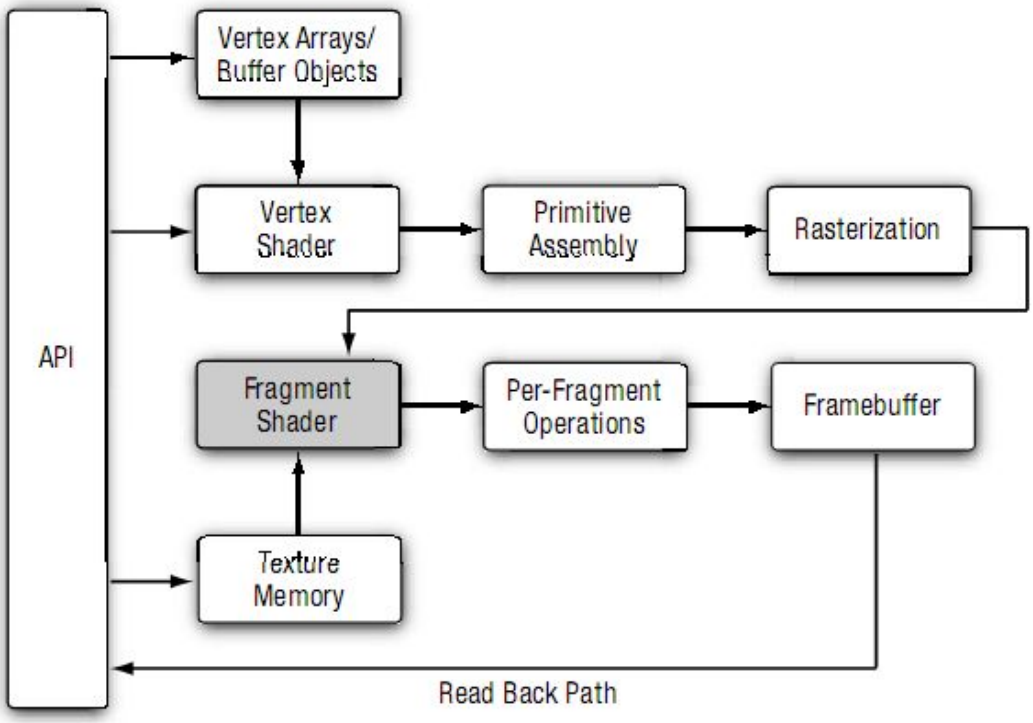


Figure 10-1 OpenGL ES 2.0 Programmable Pipeline

固定功能的片段着色器

对使用过 OpenGL ES（或者桌面版 OpenGL）先前版本，但没有使用过可编程片段管道的工作者来说，你可能熟悉固定功能片段管道。在进入片段着色器细节前，简单的讲述一下老的固定功能片段管道，将让你懂得老的固定功能管道如何匹配片段着色器。这是开始进入更先进的片段编程技术前好的开始。

OpenGL ES 1.1（或者固定管道桌面版）中，你有一些设定怎么联合各种输入到片段着色器这些方程式的限制。在固定功能管道中，本质上你有三种输入可以使用：顶点颜色一般是先前计算出来的，或者是顶点光照计算的结果；贴图颜色一般是使用基元贴图坐标绑定贴图；常量颜色一般被设定为每个贴图单元的颜色。

你能使用混合输入的方程是有限的。例如 OpenGL ES 1.1 可用的方程列于表 10-1.

表 10-1 OpenGL ES 1.1 RGB 混合行为

RGB 混合功能	方程
REPLACE	A
MODULATE	A×B
ADD	A + B
ADD_SIGNED	A + B - 0.5
INTERPOLATE	A×C + B×(1-C)
SUBTRACT	A -B
DOT3_RGB (和 DOT3_RGBA)	4×((A.r - 0.5)× (B.r - 0.5) + (A.g - 0.5)× (B.g - 0.5) + (A.b - 0.5)×(B.b - 0.5))

输入 A、B 和 C 来自输入的顶点颜色、贴图颜色、常量颜色。即使只使用了这些有限的方程，也能实现大量有趣的效果。但对于可编程，片段管道仅仅被配置成固定的方法，还是差距较远。

为什么要回顾这些，因为这将帮助懂得传统固定功能着色器技术能实现的东西。例如我们已经配置固定功能管道为单一贴图匹配，我们想去使用顶点颜色调整。固定功能管道 OpenGL ES (或 OpenGL)会使能单元贴图单元，选择混合模式为 MODULATE，设定方程输入来自顶点颜色和贴图颜色。OpenGL ES 1.1 的代码如下：

```
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE);
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB, GL_MODULATE);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_RGB, GL_PRIMARY_COLOR);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE1_RGB, GL_TEXTURE);
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_ALPHA, GL_MODULATE);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE0_ALPHA, GL_PRIMARY_COLOR);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE1_ALPHA, GL_TEXTURE);
```

这些代码配置固定功能管道去执行(A×B)即原始色（顶点颜色）颜色和贴图颜色相乘。如果这些代码对你没有什么意义，因为 OpenGL ES 2.0 没有这些东西。我们想让你知道的是片段着色器怎么匹配这些功能。在片段着色器里，计算是这样完成的。

```
precision mediump float;
uniform sampler2D s_tex0;
varying vec2 v_texCoord;
varying vec4 v_primaryColor;
void main()
{
    gl_FragColor = texture2D(s_tex0, v_texCoord) * v_primaryColor;
}
```

片段着色器执行的操作和固定功能管线是一样的。从采样器取得的贴图值和 2D 贴图坐标被使用去查找那个值。然后贴图获得的结果被 v_primaryColor 乘，它是从顶点着色器传入的变量，在这种情况下，顶点着色器传递颜色到片段着色器里。

写片段着色器执行和固定功能管线贴图联合相同的计算是可能的。这就是说，写使用很多复合体和变量计算的着色器就像固定行为一样。无论如何，本章的内容是我们转变从固定功能行为到可编程着色器，现在我们看片段着色器的细节。

片段着色器预览

片段着色器提供一个多用途可编程的着色器运行方法。片段着色器的输入有：

Varyings　　顶点着色器产生的差值数据
Uniforms　　片段着色器使用的状态
Textures　　通过采样器产生的纹理图像
Code　　片段着色器源码或二进制码，描述片段着色器将执行的操作
片段着色器的输出是通过管道的 per-fragment 操作的片段颜色，输入和输出看图 10-2.

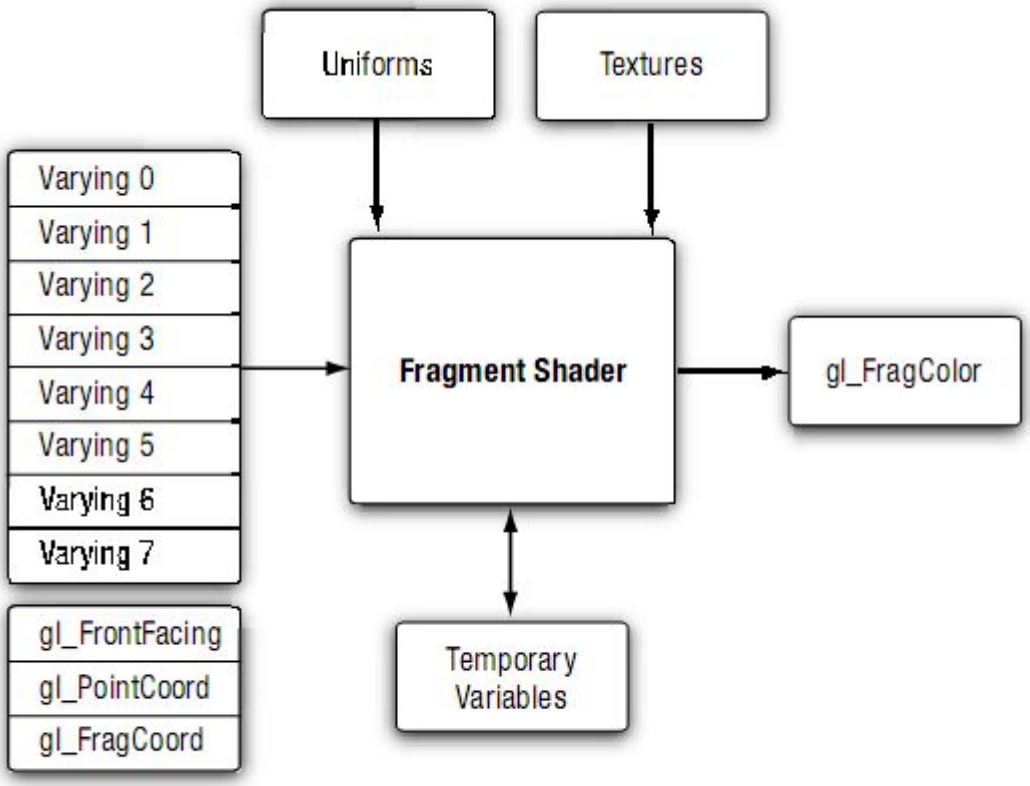


Figure 10-2　　OpenGL ES 2.0 Fragment Shader

内置的特殊变量

OpenGL ES 2.0 内置的特殊变量作为片段着色器的输入和输出。这些变量有:

gl_FragColor —gl_FragColor 被用作着色器的输出，颜色输入管道的 per-fragment 操作。如果片段着色器没有对 gl_FragColor 写数据，gl_FragColor 数据是未定义的。在着色器里不写 gl_FragColor 可能也是有合法的。例如你想渲染的仅仅是深度，能使用 glColorMask 关断颜色缓冲区。跳过写颜色缓冲区也是合法的。

gl_FragCoord —片段着色器里的只读变量，这个值容纳着窗口相关的片段坐标(x, y, z, 1/w)。有一些计算当前片段窗口坐标的算法。例如你能使用窗口坐标作为贴图获取随机噪声匹配的偏置，这个值被用于阴影匹配的旋转过滤核心。这技术被用于减少阴影匹配走样。

gl_FrontFacing —片段着色器的只读变量，如果片段是基元的前面的部分，这个布尔变量的值是 true，否则为 false。

gl_PointCoord —用于渲染点的只读变量。它在容纳着点的贴图坐标，在点光栅化阶段自动产生，范围在[0, 1]。第 13 章 OpenGL ES 2.0 先进的编程，有一个使用这个变量渲染点的例子。

内置常量

下面的内置常量对片段着色器有重大作用

```
const mediump int gl_MaxTextureImageUnits = 8;
const mediump int gl_MaxFragmentUniformVectors = 16;
const mediump int gl_MaxDrawBuffers = 1;
```

内置常量描述下面的最大值

gl_TextureImageUnits—可用的最大的贴图单元数，最小值是 8

gl_MaxFragmentUniformVectors—片段着色器里使用的最大的常量变量输入 vec4 数。支持的最小数是 16 个 vec4。可用的 vec4 数目对不同的编译器和不同的片段着色器是不同的。第章顶点着色器描述的内容对片段着色器同样适用。

gl_MaxDrawBuffers—可用的绘制缓冲区最大数。最小值是 1，支持的数超过 1，意味着编译器支持多渲染目标（MRTs），ES 2.0 不支持多 MRTs，这可能留给未来的版本支持。

内置常量的最小值必须被 OpenGL ES 2.0 编译器支持。编译器可以支持大于最小值，实际的支持值可以用下面的代码查询：

```
GLint     maxTextureImageUnits, maxFragmentUniformVectors;
glGetIntegerv(GL_MAX_TEXTURE_IMAGE_UNITS, &maxTextureImageUnits);
glGetIntegerv(GL_MAX_FRAGMENT_UNIFORM_VECTORS,
                &maxFragmentUniformVectors);
```

质量精度

在第 5 章 OpenGL E 着色器语言和第 8 章顶点着色器详细描述过。请复习这些章内容。需要提醒的是片段着色器里没有默认的精度，这意味着每个片段着色器要提供一个默认精度(或为所有变量声明提供精度质量)。

OpenGL ES 2.0 授权编译器在片段着色器里至少支持中等质量精度，但不要求高质量。片段着色器是否支持高精度取决于 GL_FRAGMENT_PRECISION_HIGH 宏是否被定义（编译器会扩展 GL_OES_fragment_precision_high 字符串）。

ES 2.0 片段着色器编程限制

第 8 章我们讨论了顶点着色器限制，和怎样写容易移植的着色器。如果你需要回顾，复习一下材料，对片段着色器有相同的限制。对片段着色器限制的唯一的不同是常量变量矩阵能够使用整型常量表达式索引。在顶点着色器里，要求编译器支持使用计算表达式值的常量变量矩阵索引。然而对片段着色器不是这样。常量变量的索引如果不使用整型常量，不保证被 ES 2.0 编译器支持。

使用着色器执行固定行为技术

现在我们给片段着色器一个总述，显示使用着色器执行几个固定行为技术例子。OpenGL ES 1.x 和桌面版 OpenGL 的固定行为管道提供的 API 可以执行多贴图混合、雾、透明度测试和使用剪切平面。所有这些在 OpenGL ES 2.0 都没有提供，但它们仍然可以使用着色器执行。下面的章节回顾那些固定行为过程提供演示这些技术的片段着色器例子。

多贴图混合

多贴图混合是片段着色器里很普遍的操作，被使用混合多个贴图映射。这个技术在很多游戏里被使用，像 Quake III，为贴图映射存储使用光传递方程预计算的光。这种映射混合片段着色器里的基本贴图和静态光。有一些使用多种贴图的例子，我们在 13 章讲述。例如贴图映射经常存储一些特殊的指数，和遮挡减弱及遮挡特殊的光贡献。很多游戏也使用法线映射，这是贴图存储每个法线信息的细节，以至于光照能够在片段着色器里执行。

而这所有在这里提到的，你现在已经了解所有需要多纹理技术来完成的 API。第 9 章，你学会怎样去装载各种贴图单元，并把它们输入到片段着色器。在片段着色器里使用各种方式混合贴图，就是采用很多操作和片段着色器里内置函数的简单事情。使用这些技术，你能实现先前 OpenGL ES 版本固定功能管线着色器做的各种可能。

使用多贴图的例子看 10-3.
这个例子装载基本的贴图和光照贴图，在正方形内混合他们。片段着色器采样程序看例 10-1。

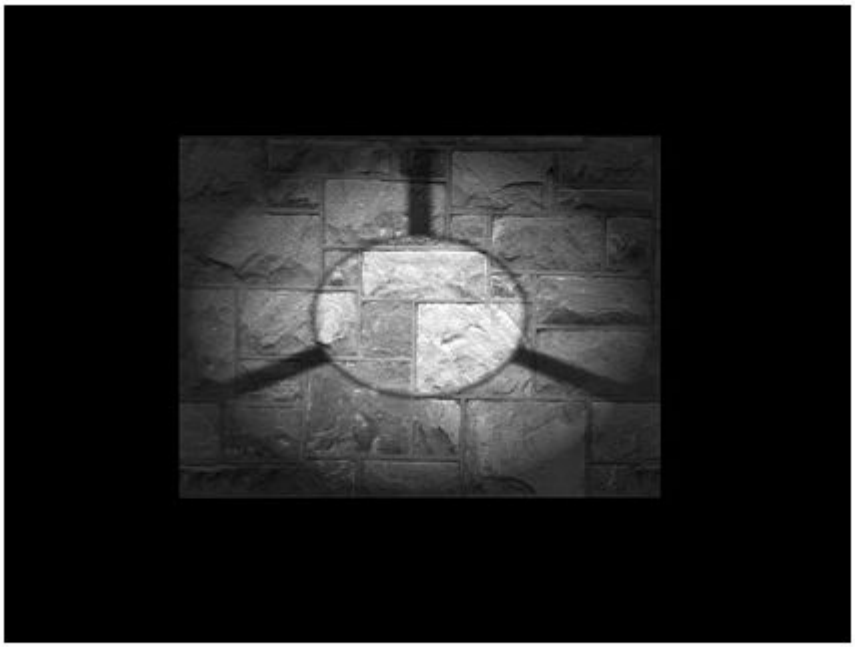


Figure 10-3 Multitextured Quad

例 10-1 多贴图片段着色器

```
precision mediump float;
varying vec2 v_texCoord;
uniform sampler2D s_baseMap;
uniform sampler2D s_lightMap;
void main()
{
    vec4 baseColor;
    vec4 lightColor;

    baseColor = texture2D(s_baseMap, v_texCoord);
    lightColor = texture2D(s_lightMap, v_texCoord);
    gl_FragColor = baseColor * (lightColor + 0.25);
}
```

片段着色器有两个采样器，一个是贴图，建立贴图单元和采样器代码如下：

```
// Bind the base map
glActiveTexture(GL_TEXTURE0);
glBindTexture(GL_TEXTURE_2D, userData->baseMapTexId);
// Set the base map sampler to texture unit 0
glUniform1i(userData->baseMapLoc, 0);
// Bind the light map
glActiveTexture(GL_TEXTURE1);
glBindTexture(GL_TEXTURE_2D, userData->lightMapTexId);
```

// Set the light map sampler to texture unit 1
glUniform1i(userData->lightMapLoc, 1);
就像你看到的，这段代码绑定独立的贴图对象到贴图单元 0 和 1。采样器被设定绑定到各自的贴图单元。例子中单一的 2 贴图坐标被使用去设定两个的匹配。在典型的光照匹配上，基本映射和光源映射应该是独立的坐标。光照映射典型应该是记录为大的独立的贴图，贴图坐标一个使用离线工具产生。
雾
一个普遍使用渲染 3D 场景的技术是应用雾。OpenGL ES 1.1 和桌面版 OpenGL，雾功能是固定行为管线提供。雾是如此流行的原因是能够被使用减少绘制远的物体和向观察者几靠近的何物体的 popping 现象。
使用可编程着色器你没有使用任何特殊方程的限制，有多种方法来计算雾。这里显示你计算线性雾。为计算雾，需要两个输入：像素到眼睛的距离和雾的颜色。为计算线性雾，我们需要物覆盖最大和最小的距离范围。
计算线性雾因子的方程为：

$$F = \frac{MaxDist - EyeDist}{MsxDist - MinDist}$$

用雾因子乘雾的颜色。颜色值被限制在[0.0, 1.0]范围内，线性插补片段着色器其它的颜色值。到眼睛的距离最好在顶点着色器计算使用变量对基元插补。
RenderMonkey 提供第 10 章线性雾的例子，显示的图像看图 10-4.

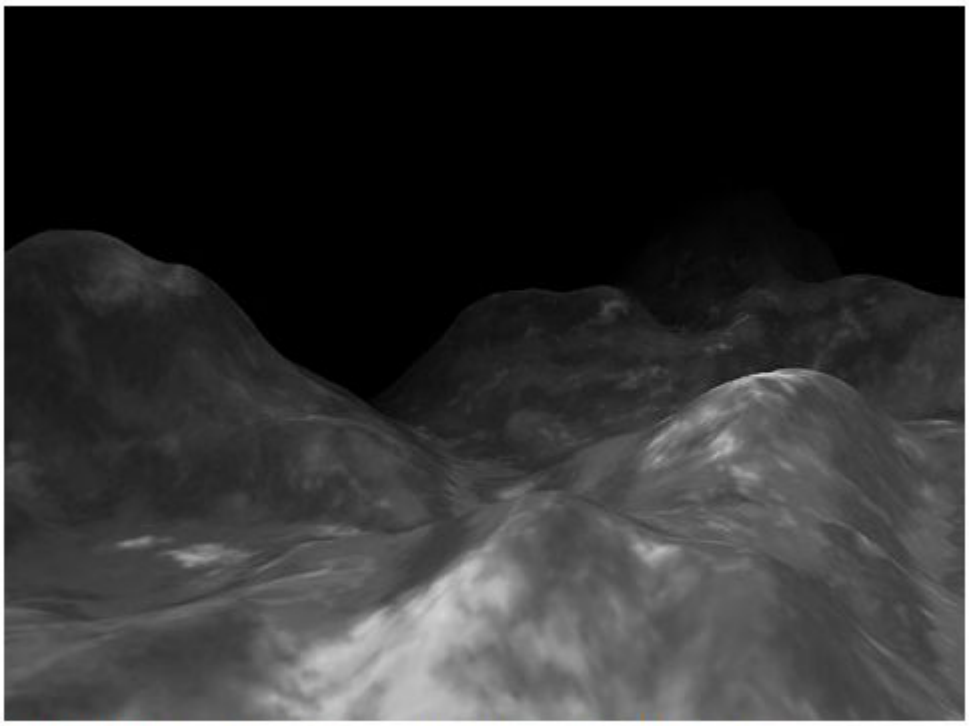


Figure 10-4 Linear Fog on Terrain in RenderMonkey

顶点着色器里计算到眼睛的距离看例 10-2.
例 10-2 顶点着色器计算到眼睛的距离
uniform mat4 matViewProjection;
uniform mat4 matView;
uniform vec4 u_eyePos;

```
attribute vec4 rm_Vertex;
attribute vec2 rm_TexCoord0;
varying vec2 v_texCoord;
varying float v_eyeDist;
void main(void)
{
    // Transform vertex to view-space
    vec4 vViewPos = matView * rm_Vertex;

    // Compute the distance to eye
    v_eyeDist = sqrt((vViewPos.x - u_eyePos.x) *
                     (vViewPos.x - u_eyePos.x) +
                     (vViewPos.y - u_eyePos.y) *
                     (vViewPos.y - u_eyePos.y) +
                     (vViewPos.z - u_eyePos.z) *
                     (vViewPos.z - u_eyePos.z) );
    gl_Position = matViewProjection * rm_Vertex;
    v_texCoord    = rm_TexCoord0.xy;
}
```

顶点着色器最重要的计算是 v_eyeDist 变量。首先顶点着色器的输入使用存储在 vViewPos 里的视图矩阵传递到视觉空间。然后从这点到 u_eyePos 常量变量的距离被计算。计算出眼睛空间里观察者到转换矩阵的距离。我们能够使用这个值在片段着色器里计算例 10-3 的雾因子。

例 10-3 片段着色器的线性雾

```
precision mediump float;
uniform vec4 u_fogColor;
uniform float u_fogMaxDist;
uniform float u_fogMinDist;
uniform sampler2D baseMap;
varying vec2 v_texCoord;
varying float v_eyeDist;
float computeLinearFogFactor()
{
    float factor;

    // Compute linear fog equation
    factor = (u_fogMaxDist - v_eyeDist) /
             (u_fogMaxDist - u_fogMinDist);

    // Clamp in the [0,1] range
    factor = clamp(factor, 0.0, 1.0);

    return factor;
}
```

```
void main(void)
{
    float fogFactor = computeLinearFogFactor();
    vec4    fogColor    = fogFactor * u_fogColor;
    vec4 baseColor = texture2D( baseMap, v_texCoord );
        // Compute final color as a lerp with fog factor
    gl_FragColor = baseColor * fogFactor +
                        fogColor    * (1.0 - fogFactor);
}
```

片段着色器里 computeLinearFogFactor()函数计算线性雾方程。最小和最大的雾距离被存储在常量变量里，在顶点着色器里计算的眼睛距离被插补后用来计算雾因子。雾因子被使用执行基本贴图颜色和雾颜色的插补。最终产生的线性雾能够通过改变 uniform 来调整距离和颜色。

注意灵活的可编程片段着色器容易增加其它方式来计算雾。例如通过改变雾方程计算指数雾。而且除了使用基于眼睛的距离，你能使用基于地面的距离来计算地面的雾。只通过修改上面提供的代码一小部分就可实现。

透明度测试（使用丢弃）

使用 3D 应用一个普遍的影响是在某些片段绘制透明的基元。这看起来像绘制网状的场景。表现这样一个场景将用到大量的基元。另一种方法是在贴图时使用一个存储有遮挡值的几何体，这个值指示那些纹理应该是透明的。例如可以使用 RGBA 贴图格式存储网状场景，RGB 代表贴图场景的颜色，A 代表纹理的遮挡值是否透明。那么在片段着色器里使用一或两个三角形和像素遮挡将很容易渲染场景。

在传统的固定功能行为渲染，这个效果使用透明度测试完成。透明度测试允许你指定一个对比值，如果片段着色器的透明度值对比一个参考值失败，片段着色器将退出。就是说片段着色器失败透明度测试，片段着色器将不渲染。OpenGL ES 2.0 里没有固定行为的透明度测试，但同样的效果能够实现使用 discard 关键字。

第 10 章，RenderMonkey 的 RM_AlphaTest 例子显示片段着色器透明度测试的例子，结果看图 10-5.

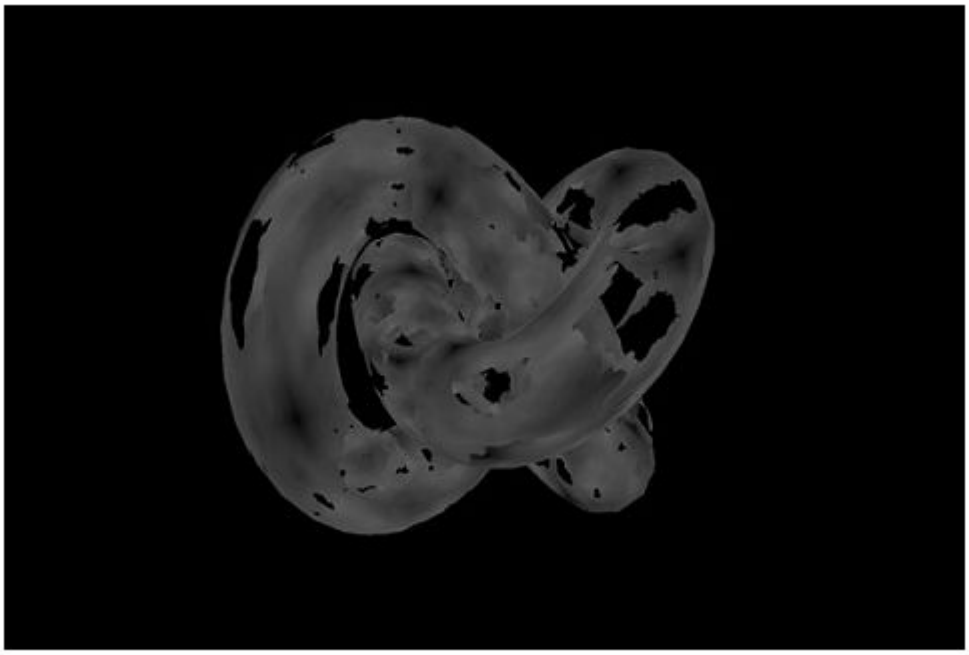


Figure 10-5 Alpha Test Using Discard

片段着色器源码看例 10-4

例 10-4 片段着色器里使用 Discard 的透明度测试

```
precision mediump float;
uniform sampler2D baseMap;
varying vec2 v_texCoord;
void main(void)
{
    vec4 baseColor = texture2D(baseMap, v_texCoord);

    if(baseColor.a < 0.25)
    {
        discard;
    }
    else
    {
        gl_FragColor = baseColor;
    }
}
```

在这个片段着色器里，贴图是 4 位的 RGBA 格式。Alpha 位被用来进行透明度测试。Alpha 颜色是 0.25 或更少，使用 discard 的片段不绘制。别的地方频道被使用贴图颜色绘制。这个技术能被提供改变对比方式或 alpha 参考值来增强。

使用剪切面

第 7 章，基元装配和光栅化，所有的基于那被六个面的投影平截头体剪切。然而有时使用者可能要用一个剪切平面去剪切。有这些情况可能用到剪切平面。产生镜面反射时，你需要剪切一个几何物体的反射面，渲染成不显示的贴图。渲染这个贴图时，你需要剪切几何体成反射面，这要求使用平面剪切。

OpenGL ES 1.1 中，提供剪切平面方程，剪切被硬件自动匹配。OpenGL ES 2.0 也能完成这个工作，但需要你自己处理。关键是使用 discard 关键字。

在讲述使用剪切平面前，先看看平面如何定义：

Ax + By + Cz + D = 0

矢量(A, B, C)是平面法线，D 从原点沿矢量到平面的距离。计算一个点是否应该被剪切平面剪切，需要计算点 P 到平面的距离，使用的方程是：

Dist = (A×P.x) + (B×P.y) + (C×P.z) + D

如果距离小于 0，这个点在平面内，被剪切；如果距离大于等于 0 不剪切。注意平面方程和 P 点必须是同一坐标系。第 10 章 RM_ClipPlane 的 RenderMonkey 例子显示图 10-6 中。

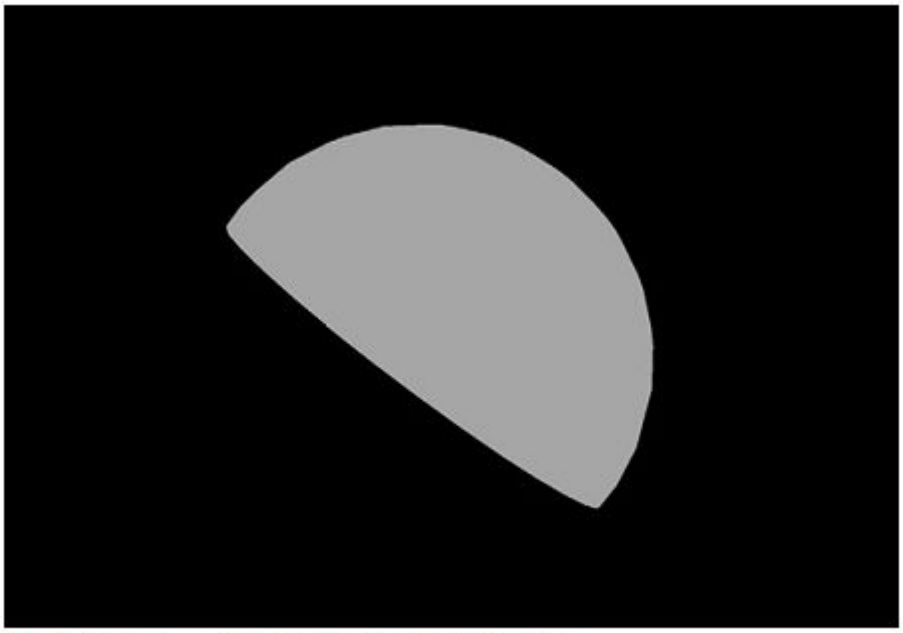


Figure 10-6 User Clip Plane Example

第一件事着色器做的是计算平面的距离。这应该在顶点着色器（使用变量输入）或片段着色器里完成。在顶点着色器里计算将比片段着色器里计算更好。例 10-5 显示在顶点着色器里计算到平面距离。

例 10-5 在顶点着色器里使用剪切平面

```
uniform vec4 u_clipPlane;
uniform mat4 matViewProjection;
attribute vec4 rm_Vertex;
varying float u_clipDist;
void main(void)
{
    // Compute the distance between the vertex and the clip plane
    u_clipDist = dot(rm_Vertex.xyz, u_clipPlane.xyz) +
                    u_clipPlane.w;
    gl_Position = matViewProjection * rm_Vertex;
}
```

u_clipPlaneu niform 容纳着剪切平面的方程。u_clipDist 变量存储着计算的距离。这些值输入到片段着色器里，使用插值距离判断片段着色器是否应该被剪切看例子 10-6.

例 10-6 片段着色器里使用剪切平面

```
precision mediump float;
varying float u_clipDist;
void main(void)
{
    // Reject fragments behind the clip plane
    if(u_clipDist < 0.0)
        discard;

    gl_FragColor = vec4(0.5, 0.5, 1.0, 0.0);
}
```

你看到，u_clipDist 值为负，以为片段在剪切平面内，被抛弃掉。别的被使用。这个例子显示了使用用户剪切平面的计算。你能增加用户剪切平面，通过计算剪切距离和进行抛弃测试。

片段着色器操作

本章讨论的操作应用于帧缓冲区或是 OpenGL ES 2.0 片段管道中执行片段着色器操作后的个体碎片，片段着色器的输出是片段的颜色和深度值。片段着色器执行后的操作能影响可见和最终像素的颜色是：

剪切盒测试
模板缓冲区测试
深度缓冲区测试
多重采样抗锯齿
混合
抖动

片段测试和操作帧缓冲区的内容显示在图 11-1.

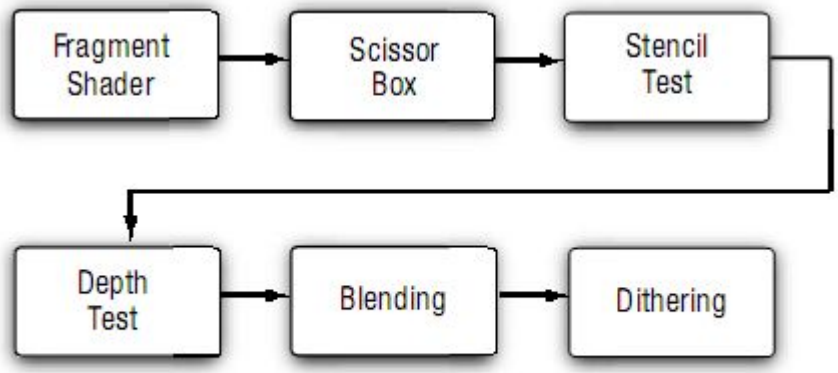


Figure 11-1 The Post-Shader Fragment Pipeline

你可能已经注意到，图中的阶段没有多重采样。多重采样技术是子碎片操作等级的副品。我们在后面描述多重采样对片段过程的影响。

本章讨论从写缓冲区到读缓冲区。

缓冲区

OpenGL ES 支持三种类型的缓冲区，在缓冲区里它们为每个像素存储不同的数据。

颜色缓冲区
深度缓冲区
模板缓冲区

缓冲区的尺寸——参考缓冲区深度（不要和深度缓冲区混淆）——能用来存储单个像素信息的位数。例如，颜色缓冲区有三个成分组成：红、绿和蓝组成，还有一个可选的透明度成员。颜色缓冲区的深度是所有它成员的位数的总和。对深度和模板缓冲区只有一个单一的像素位深。例如深度缓冲区可能有每像素 16 位。缓冲区位深度是所有成员位深度的和。一般来说，深度缓冲区每像素有 16 位的是 5 位红色和蓝色，六位绿色（人的可见光绿色比和蓝色更敏感）如果是 32 位，平等的分给 RGBA 缓冲区。

颜色缓冲区可能是双缓冲区，它包括两缓冲区：一个是输出设备的显示（经常是监视器或 LCD 显示器）称为前缓冲区，另一个对观察者是隐藏的，单被使用于构造下一个图像的显示，称作后缓冲区。使用双缓冲区应用时，图像被绘制到后缓冲区，然后交换前后缓冲区去显示下一幅图像。缓冲区的交换和显示设备的更新周期经常是同步的，这产生连续平滑活动画面的错觉，有关双缓冲区讨论看第 3 章 EGL 介绍。

虽然每个 EGL 配置有颜色缓冲区、深度缓冲区和可选的模板缓冲区，然而每个 EGL 必须提供包括三个缓冲区的配置，深度缓冲区至少 16 位深度，模板缓冲区至少 8 位。

要求附加的缓冲区

包括深度缓冲区、模板缓冲区伴随着颜色缓冲区，当你为你的 EGL 指定配置属性时，你需要请求它们。回忆第 3 章，你输入一系列属性值到 EGL 指定你渲染需要的平面类型。包括深度缓冲区和颜色缓冲区你能使用你需要的位深度指定属性列表的 EGL_DEPTH_SIZE。你能使用 EGL_STENCIL_SIZE 指定你要求的模板缓冲区位数。

我们的 esUtil 库，简化了那些操作，你仅仅需要指出你希望和颜色缓冲区一起使用的缓冲区，库函数将做剩余的工作（要求最大的缓冲区尺寸）。使用库时，你必须在你的函数 esCreateWindow 中加入 ES_WINDOW_DEPTH and ES_WINDOW_STENCIL，例如：

```
esCreateWindow(&esContext, "Application Name",
                window_width, window_height,
                ES_WINDOW_RGB | ES_WINDOW_DEPTH | ES_WINDOW_STENCIL);
```

创建缓冲区

OpenGL ES 是一个交互式渲染系统，假设每帧的开始，你必须初始化所有缓冲区到初始默认值。调用 glClear 函数清除缓冲区内容，参数 mask 指定清除的缓冲区。

```
void    glClear(GLbitfield mask);
```

mask　指定需要清除的缓冲区，被下面代表各种 OpenGL ES 缓冲区的遮挡位组成:
GL_COLOR_BUFFER_BIT,
GL_DEPTH_BUFFER_BIT, GL_STENCIL_BUFFER_BIT

你可能不要求清除每一个缓冲区，不在同时清除它们。但如果你想同时清除所有的缓冲区。每帧调用一次 glClear 可以得到最好的执行。

当你要求清除缓冲区时，缓冲区的默认值被使用。每个缓冲区你可以指定默认的清除颜色。

```
void    glClearColor(GLclampf red, GLclampf green,
                     GLclampf blue, GLclampf alpha);
```

red, green, blue, alpha　当使用 GL_COLOR_BUFFER_BIT 作为 glClear 的遮挡位，指定颜色缓冲区内所有将被初始化像素的颜色值（范围[0,1]）。

```
void    glClearDepthf(GLclampf depth);
```

depth　当使用 GL_COLOR_BUFFER_BIT 作为 glClear 的遮挡位，指定深度缓冲区内所有将被初始化像素的深度值（范围[0,1]）。

```
void    glClearStencil(GLint s);
```

s　指定模板值（范围 [0,2^n-1]，n 是模板缓冲区可用的位数值）当使用 GL_COLOR_BUFFER_BIT 作为 glClear 的遮挡位，指定深度缓冲区内所有将被初始化像素的深度值

控制帧缓冲区写入遮挡

你能控制每个缓冲区甚至是它的组成部分，对颜色缓冲区，能使用 mask.控制缓冲区是否可写。像素值被写入缓冲区前，缓冲区的 mask.指定缓冲区是否可用。

对颜色缓冲区，glColorMask 指定颜色缓冲区的组成里那部分可被更新。如果 mask 被设为 GL_FALSE，这个组成部分不能更新，默认值是可用的。

```
void    glColorMask(GLboolean red, GLboolean green,
                    GLboolean blue, GLboolean alpha);
```

red, green, 指定渲染时颜色缓冲区特定的颜色组成是否可以修改

blue,
alpha

一般的，如果深度缓冲区可用，使用 GL_TRUE 或 GL_FALSE 为参数，调用 glDepthMask 指定对深度缓冲区的写入。

当渲染透明物体时，经常去不使能深度缓冲区。首先你使用深度缓冲区使能渲染所有场景中不透明的物体，确保所有的不透明物体深度值被正确分类，深度缓冲区存储着场景的合适的深度信息。然后，调用 glDepthMask(设定为 GL_FALSE)不使能深度缓冲区，渲染透明物体。深度缓冲区不能写的时候，它仍然可以读，并且可以进行比较。使用纠正过得深度缓冲区值，允许透明物体被不透明物体变暗，因为没有修改深度值，不透明物体将透明物体变暗。

void glDepthMask(GLboolean depth);
depth 指定深度缓冲区是否可以修改

最后，你调用 glStencilMask 不使能对模板缓冲区写，但对比 glColorMask 或 glDepthMask，你使用 mask 指定模板缓冲区哪一位可写。

void glStencilMask(GLuint mask);
mask 指定模板缓冲区中像素被修改的遮挡位 (范围 $[0,2^n-1]$,n 是模板缓冲区位数)

glStencilMaskSeparate 允许你基于基元面上顶点的顺序设定模板遮挡位。这允许对基元的前和后面设定不同的模板遮挡位。调用 glStencilMaskSeparate(GL_FRONT_AND_BACK, mask)和 glStencilMask 调用是相同的，它为多边形的前后面设定遮挡位。

void glStencilMaskSeparate(GLenum face, GLuint mask);
face 基于渲染基元表面的顶点顺序，指定使用的模板遮挡，可用值有
GL_FRONT, GL_BACK, 和 GL_FRONT_AND_BACK
mask 指定模板缓冲区中某个面的像素遮挡位(范围$[0,2^n]$, n 是模板缓冲区位数)

片段测试和操作

下面章节描述能被 OpenGL ES 片段着色器使用的各种测试。默认状态，所有的片段着色器测试和操作是不使能的。片段着色器按照要求的顺序把片段转化为像素写到帧缓冲区中。使能各种片段测试操作，能够选择哪些片段转变成像素影响到最终的图像输出。

每个片段测试可以使用 glEnable 独立使能，参数列表看表 11-1。

Table11-1 片段测试使能符号

使能符号	描述
GL_DEPTH_TEST	片段深度测试控制
GL_STENCIL_TEST	片段模板测试控制
GL_BLEND	颜色缓冲区存储的片段颜色混合测试控制
GL_DITHER	写入颜色缓冲区前片段颜色抖动控制
GL_SAMPLE_COVERAGE	估算采样平均值控制
GL_SAMPLE_ALPHA_TO_COVERAGE	估算采样平均值使用的采样透明度控制

剪切测试

剪切测试通过指定一个矩形区域限制写入帧缓冲区中的像素，提供附加的剪切操作。使用剪切分两步，首先使用 glScissor 指定矩形区域。

void glScissor(GLint x, GLint y, GLsizei width,
GLsizei height);
x, y 指定视图坐标系中剪切矩阵左和下的边。

width　以像素为单位指定剪切盒子的宽度
height　以像素为单位指定剪切盒子的高度
指定矩形区域后，调用 glEnable(GL_SCISSOR_TEST)使能剪切。所有的渲染操作，包括清除可视区，都被剪切操作限制。

一般的剪切区域是可视区的子区域，但不要求两个区域实际交叉。

模板缓冲区测试

下一个片段可用的操作是模板测试。模板缓冲区每个像素都可以设置遮挡，控制着像素值是否可以被更新。模板测试被应用程序使能或关闭。

使用模板测试要两步。第一步初始化模板缓冲区每个像素的遮挡位，通过渲染几何图形和指定模板缓冲区被如何更新。第二步，使用那些值控制子区域对颜色缓冲区的渲染。所有情况下，你指定的参数被用于模板测试。

模板测试是重要的位测试，例如你能在 C 语言里决定那个位被设定为遮挡。模板行为的控制着操作者和测试值函数是：

void　glStencilFunc(GLenum func, GLint ref, GLuint mask);
void　glStencilFuncSeparate(GLenum face, GLenum func,
　　GLint ref, GLuint mask);
face　指定面与模板的联系，可用值是：
　　GL_FRONT, GL_BACK, and GL_FRONT_AND_BACK
Func　指定模板测试的对比行为，可用值是：
　　GL_EQUAL, GL_NOTEQUAL, GL_LESS, GL_GREATER,
　　GL_LEQUAL, GL_GEQUAL, GL_ALWAYS, and GL_NEVER
ref　指定模板测试的对比值
mask　指定遮挡位和模板缓冲区中的值按位与，形成参考值。

模板测试允许最后的控制，遮挡参数被使用去选择是否被用于测试。选择哪些位后和操作者提供的参考值对比。例如指定模板缓冲区的低 3 位和 2 对比，应写成 glStencilFunc(GL_EQUAL, 2, 0x7);这样就使能了测试。

模板测试完成后，你需要让 OpenGL ES 2.0 知道如何处理模板缓冲区中的值。事实上，修改模板缓冲区中的值不但依靠模板测试，还依靠深度测试的结果（后面章节谈论）。联合深度测试和模板测试，有三种可能的片段输出结果。

1. 模板测试失败，没有更多的测试被用于片段（例如深度测试）
2. 通过了模板测试，深度测试失败
3. 模板测试和深度测试都完成

每一种可能的输出将影响模板缓冲区中的值。glStencilOp 和 glStencilOpSeparate 函数控制着模板缓冲区值对测试输出采用的实际操作。对模板值可能的操作显示在表 11-2 里。

表 11-2 模板操作

模板参数	描述
GL_ZERO	设定模板值为 0
GL_REPLACE	使用 glStencilFunc 或 glStencilFuncSeparate 指定的值取代当前模板值
GL_INCR, GL_DECR	增加或减小模板值，模板值被限定到 0 到 2^n，n 是模板缓冲区位数
GL_INCR_WRAP,	增加或减小模板值，但模板值被限定在最大（到最大值导致新德模板值是 0）或最小（到最小到 0 导致新德模板值最大的模板值）

GL_DECR_WRAP

GL_KEEP 不改变当前模板值

GL_INVERT 模板缓冲区中的值安位取反

void glStencilOp(GLenum sfail, GLenum zfail, GLenum zpass);

void glStencilOpSeparate(GLenum face, GLenum sfail,GLenum zfail, GLenum zpass);

face 根据提供的模板函数，指定面，可用值是：GL_FRONT, GL_BACK, and GL_FRONT_AND_BACK

sfail 如果片段的模板测试失败，指定对模板位的操作，可用值是：GL_KEEP,GL_ZERO, GL_REPLACE, GL_INCR, GL_DECR, GL_INCR_WRAP,GL_DECR_WRAP, 和 GL_INVERT

zfail 指定片段通过了模板测试，但深度测试失败的行为

zpass 指定片段通过了模板和深度测试的行为

下面的例子显示使用 glStencilFunc 和 glStencilOp 控制视窗各部分渲染

```
GLfloat vVertices[] =
{
    -0.75f,  0.25f,  0.50f, // Quad #0
    -0.25f,  0.25f,  0.50f,
    -0.25f,  0.75f,  0.50f,
    -0.75f,  0.75f,  0.50f,
     0.25f,  0.25f,  0.90f, // Quad #1
     0.75f,  0.25f,  0.90f,
     0.75f,  0.75f,  0.90f,
     0.25f,  0.75f,  0.90f,
    -0.75f, -0.75f,  0.50f, // Quad #2
    -0.25f, -0.75f,  0.50f,
    -0.25f, -0.25f,  0.50f,
    -0.75f, -0.25f,  0.50f,
     0.25f, -0.75f,  0.50f, // Quad #3
     0.75f, -0.75f,  0.50f,
     0.75f, -0.25f,  0.50f,
     0.25f, -0.25f,  0.50f,
    -1.00f, -1.00f,  0.00f, // Big Quad
     1.00f, -1.00f,  0.00f,
     1.00f,  1.00f,  0.00f,
    -1.00f,  1.00f,  0.00f
};
GLubyte indices[][6] =
{
    {  0,  1,  2,  0,  2,  3 }, // Quad #0
    {  4,  5,  6,  4,  6,  7 }, // Quad #1
    {  8,  9, 10,  8, 10, 11 }, // Quad #2
    { 12, 13, 14, 12, 14, 15 }, // Quad #3
    { 16, 17, 18, 16, 18, 19 }  // Big Quad
```

```
    };

#define NumTests    4
    GLfloat    colors[NumTests][4] =
    {
        { 1.0f, 0.0f, 0.0f, 1.0f },
        { 0.0f, 1.0f, 0.0f, 1.0f },
        { 0.0f, 0.0f, 1.0f, 1.0f },
        { 1.0f, 1.0f, 0.0f, 0.0f }
    };
    GLint      numStencilBits;
    GLuint     stencilValues[NumTests] =
    {
        0x7, // Result of test 0
        0x0, // Result of test 1
        0x2, // Result of test 2
        0xff // Result of test 3. We need to fill this
             // value in a run-time
    };
    // Set the viewport
    glViewport(0, 0, esContext->width, esContext->height);

    // Clear the color, depth, and stencil buffers. At this
    // point, the stencil buffer will be 0x1 for all pixels
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT |
            GL_STENCIL_BUFFER_BIT);
    // Use the program object
    glUseProgram(userData->programObject);
    // Load the vertex position
    glVertexAttribPointer(userData->positionLoc, 3, GL_FLOAT,
                          GL_FALSE, 0, vVertices);
    glEnableVertexAttribArray(userData->positionLoc);
    // Test 0:
    //
    // Initialize upper-left region. In this case, the stencil-
    // buffer values will be replaced because the stencil test
    // for the rendered pixels will fail the stencil test, which is
 //
    //          ref     mask     stencil    mask
    //        ( 0x7 & 0x3 ) < ( 0x1 & 0x7 )
    //
    // The value in the stencil buffer for these pixels will
    // be 0x7.
    //
```

```
glStencilFunc(GL_LESS, 0x7, 0x3);
glStencilOp(GL_REPLACE, GL_DECR, GL_DECR);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, indices[0]);
// Test 1:
//
// Initialize the upper right region. Here, we'll decrement
// the stencil-buffer values where the stencil test passes
// but the depth test fails. The stencil test is
//
//             ref    mask       stencil    mask
//          ( 0x3 & 0x3 ) > ( 0x1 & 0x3 )
//
//       but where the geometry fails the depth test.    The
//       stencil values for these pixels will be 0x0.
//
glStencilFunc(GL_GREATER, 0x3, 0x3);
glStencilOp(GL_KEEP, GL_DECR, GL_KEEP);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, indices[1]);
// Test 2:
//
// Initialize the lower left region. Here we'll increment
// (with saturation) the stencil value where both the
// stencil and depth tests pass. The stencil test for
// these pixels will be
//
//             ref    mask       stencil    mask
//          ( 0x1 & 0x3 ) == ( 0x1 & 0x3 )
//
// The stencil values for these pixels will be 0x2.
//
glStencilFunc(GL_EQUAL, 0x1, 0x3);
glStencilOp(GL_KEEP, GL_INCR, GL_INCR);
glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, indices[2]);
// Test 3:
//
// Finally, initialize the lower right region. We'll invert
// the stencil value where the stencil tests fails. The
// stencil test for these pixels will be
//
//             ref     mask       stencil    mask
//          ( 0x2 & 0x1 ) == ( 0x1 & 0x1 )
//
// The stencil value here will be set to ~((2^s-1) & 0x1),
// (with the 0x1 being from the stencil clear value),
```

```
    // where 's' is the number of bits in the stencil buffer
    //
    glStencilFunc(GL_EQUAL, 0x2, 0x1);
    glStencilOp(GL_INVERT, GL_KEEP, GL_KEEP);
    glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, indices[3]);

    // As we don't know at compile time how many stencil bits are
    // present, we'll query, and update the correct value in the
    // stencilValues arrays for the fourth tests. We'll use this
    // value later in rendering.
    glGetIntegerv(GL_STENCIL_BITS, &numStencilBits);

    stencilValues[3] = ~(((1 << numStencilBits) - 1) & 0x1) & 0xff;
    // Use the stencil buffer for controlling where rendering will
    // occur. We disable writing to the stencil buffer so we can
    // test against them without modifying the values we generated.
    glStencilMask(0x0);

    for(i = 0; i < NumTests; ++i)
    {
        glStencilFunc(GL_EQUAL, stencilValues[i], 0xff);
        glUniform4fv(userData->colorLoc, 1, colors[i]);
        glDrawElements(GL_TRIANGLES, 6, GL_UNSIGNED_BYTE, indices[4]);
    }
```

深度缓冲区测试

深度缓冲区常常被用于隐藏表面的移动。它常常被用于保持最靠近的物体对渲染表面每个像素观察点的距离，对每一个新输入的片段，使用存储的观察点的值对比距离。默认情况，如果新输入片段的值小于存储在深度缓冲区的值（意味着它更靠近观察点），输入片段的值取代深度缓冲区的值，然后它的颜色值取代颜色缓冲区中的值。这是标准的使用深度缓冲区的方法。如果你想这样使用，你需要请求一个深度缓冲区，当你创建窗口时，使用参数 GL_DEPTH_TEST 调用 glEnable 使能深度缓冲区。如果没有深度缓冲区联系到颜色缓冲区，深度缓冲区测试总是通过。

经常，仅仅有一个使用深度缓冲区方法。你能调用 glDepthFunc 修改深度缓冲区对比操作。

void glDepthFunc(GLenum func)

func 指定对比行为的深度值，可以使：GL_LESS, GL_GREATER, GL_LEQUAL, GL_GEQUAL,GL_EQUAL, GL_NOTEQUAL, GL_ALWAYS, 或 GL_NEVER

混合

本届谈论像素颜色混合。一旦片段通过所有使能的片段测试，它的颜色将混合那已经出现在片段像素位置上的颜色。两个以上混合前，它们使用指定的混合操作乘混合因子。使用方程是：

$C_{final} = f_{source} C_{source}$ op $f_{destingation} C_{destination}$

$f_{source} C_{source}$ 是输入片段的缩放因子和颜色。$f_{destingation}$ 和 $C_{destination}$ 是像素的缩放因子和颜色。Op 是混合缩放值的操作。

缩放因子通过 glBlendFunc 或 glBlendFuncSeparate 指定

void glBlendFunc(GLenum sfactor, GLenum dfactor);

sfactor 为输入片段指定抖动系数

dfactor 为最终像素指定抖动系数

void glBlendFuncSeparate(GLenum srcRGB, GLenum dstRGB,
GLenum srcAlpha, GLenum dstAlpha);

srcRGB 为输入片段的颜色 red, green, 和 blue 成分指定抖动系数

dstRGB 为最终像素的颜色 red, green, 和 blue 成分指定抖动系数

srcAlpha 为输入片段的透明度值指定抖动系数

dstAlpha 为最终像素的透明度值指定抖动系数

可能的混合系数显示在表 11-3

表 11-3 混合行为

Blending Coefficient Enum	RGB Blending Factors	Alpha Blending Factor
GL_ZERO	(0, 0, 0)	0
GL_ONE	(1, 1, 1)	1
GL_SRC_COLOR	(R_s, G_s, B_s)	A_s
GL_ONE_MINUS_SRC_COLOR	$(1 - R_s, 1 - G_s, 1 - B_s)$	$1 - A_s$
GL_SRC_ALPHA	(A_s, A_s, A_s)	A_s
GL_ONE_MINUS_SRC_ALPHA	$(1 - A_s, 1 - A_s, 1 - A_s)$	$1 - A_s$
GL_DST_COLOR	(R_d, G_d, B_d)	A_d
GL_ONE_MINUS_DST_COLOR	$(1 - R_d, 1 - G_d, 1 - B_d)$	$1 - A_d$
GL_DST_ALPHA	(A_d, A_d, A_d)	A_d
GL_ONE_MINUS_DST_ALPHA	$(1 - A_d, 1 - A_d, 1 - A_d)$	$1 - A_d$
GL_CONSTANT_COLOR	(R_c, G_c, B_c)	A_c
GL_ONE_MINUS_CONSTANT_COLOR	$(1 - R_c, 1 - G_c, 1 - B_c)$	$1 - A_c$
GL_CONSTANT_ALPHA	(A_c, A_c, A_c)	A_c
GL_ONE_MINUS_CONSTANT_ALPHA	$(1 - A_c, 1 - A_c, 1 - A_c)$	$1 - A_c$
GL_SRC_ALPHA_SATURATE	$\min(A_s, 1 - A_d)$	1

表 11-3 (R_s, G_s, B_s, A_s)是输入片段颜色的组成部分，(R_d, G_d , B_d, A_d)是颜色缓冲区存储的像素颜色，(R_c,G_c,B_c,A_c)是通过 glBlendColor 设定的颜色常量。使用 GL_SRC_ALHPA_SATURATE 的例子，仅仅计算的最小被应用到源颜色。

void glBlendColor(GLclampf red, GLclampf green,
GLclampf blue, GLclampf alpha);

red, green, 为常量抖动颜色指定组成值

blue,

alpha
一旦输入的片段和像素颜色乘以它们各自的缩放因子，它们使用 glBlendEquation 或 glBlendEquationSeparate 执行指定的操作。默认使用 GL_FUNC_ADD，混合加运算操作。GL_FUNC_SUBTRACT 操作从输入片段值里减去帧缓冲区的颜色因子。同样的，GL_FUNC_REVERSE_SUBTRACT 颠倒混合方程，使用当前像素颜色减去输入片段颜色。

void glBlendEquation(GLenum mode);

mode 指定抖动操作，可用值是：GL_FUNC_ADD,GL_FUNC_SUBTRACT, 或 GL_FUNC_REVERSE_SUBTRACT

void glBlendEquationSeparate(GLenum modeRGB,
GLenum modeAlpha);

modeRGB 为 red, green, and blue 成分指定抖动操作

modeAlpha 指定透明度成分抖动操作

抖动

对一个帧缓冲区只有有限可用颜色的系统，我们能使用抖动去模仿更大的优势深度。抖动算法处理颜色让图像看起来好像真实存在有更多可用色。OpenGL ES 2.0 没有指定什么抖动算法被支持，这个技术也是非常依赖实现的。

你的应用对抖动仅仅的控制是是否在最后的像素上应用它，使用 glEnable 或 glDisable 控制管道是否使用抖动。

多重采样抗锯齿

抗锯齿是一个提高分离像素、减少人工痕迹提高图像质量的重要技术，OpenGL ES 2.0 中几何体基元在渲染的光栅化阶段，变成一个网格，它们的边缘在这个过程可能变得畸形。我们看到监视器上的楼梯的对角线就是如此。

有各种技术可以用来减少这种走样影响。OpenGL ES 2.0 提供的叫 multisampling，多重采样把每个像素变成一系列采样，它们中的每一个在光栅化阶段被认为是最小的像素。也就是说，当几何图形基元被渲染时，他们被渲染进入帧缓冲区真实的数据比真实显示的表面更多。每一个采样，都有它自己的颜色、深度和模板值。这些值被保留到图像准备去显示。当混合最后图像时，采用值被转换成最后的图像颜色。做这个特殊的过程是为了使用每个采用的颜色信息，OpenGL ES 2.也多少个采样在光栅化阶段发生的信息。像素的每个采样被认为是样本覆盖遮挡的一位。使用覆盖遮挡，我们能控制最后的像素转换。每一个 OpenGL ES 2.0 的创建的渲染面，甚至是一个像素都使用多重采样。

多重采样可以打开和关断（使用 glEnable 和 glDisable），有多个选项可以控制采用转换值。首先，你使用使能 GL_SAMPLE_ALPHA_TO_COVERAGE 决定采样的透明度将被用于覆盖。这种情况下，几何基元覆盖一个采样，片段输入的透明度值被用来决定附加的覆盖遮挡，使用片段采样按位 AND 计算遮挡位值，计算的值直接取代原始值，具体的采样计算和编译器工具有关。（In this mode, if the geometric primitive covers a sample, the alpha value of incoming fragment is used to determine an additional sample coverage mask computed that is bitwise ANDed into the coverage mask that is computed using the samples of the fragment.）（不懂，翻译不了）新输入的值替代原始的直接产生采样覆盖值。采样计算依靠编译工具。

另外你能指定 GL_SAMPLE_COVERAGE，它使用片段（可能被先前的操作列表容易的修改）覆盖值按位 AND（与）glSampleCoverage 指定的值。使用 glSampleCoverage 指定的值被用于产生一个编译工具指定的覆盖值，包括一个反转标志、覆盖值的按位反转。使用反转的标志位，可能创建两个透明的遮挡，不使用完整的不同的采样设置。

void glSampleCoverage(GLfloat value, GLboolean invert);

value 指定一个在[0, 1] 范围内的值，被转变成采样遮挡位。根据设定的这个值，设定一

个同比例的遮挡位。

invert　指定遮挡位后，所有指定的遮挡位被反转。

多重采样帮助减少场景中的锯齿，但也可能产生难看的人工痕迹。这常常是因为采样的像素位置引起的。使用重心采样有助于纠正这个问题，不幸的是 OpenGL ES 2.0 现在还不支持这个特征。

缓冲区读写像素

如果你想保存原先渲染的图像，你能从颜色缓冲区读回像素值，但不能从深度和模板缓冲区读回数据。调用 glReadPixels，读颜色缓冲区中的数据存储到你先前指定的矩阵中。

```
void    glReadPixels(GLint x, GLint y, GLsizei width,
                     GLsizei height, GLenum format,
                     GLenum type, void *pixels);
```

x, y　指定从颜色缓冲区读回的像素矩形的视图坐标系的左和下边。

width, height　指定从颜色缓冲区读回的像素矩形的规模

format　指定返回的像素格式，可用值是: GL_RGB，返回值可以通过 GL_IMPLEMENTATION_COLOR_READ_FORMAT 查询，这是特殊的像素格式。

type　指定返回的像素数据类型，可用值是: GL_UNSIGNED_BYTE，返回值可以通过 GL_IMPLEMENTATION_COLOR_READ_TYPE 查询，这是特殊的像素类型

pixels　是一个连续的数据矩阵，包含着使用 glReadPixels 返回的颜色缓冲区中值

除了像素格式(GL_RGB)和类型(GL_UNSIGNED_BYTE)，你注意到返回的值依靠你使用的编译工具支持的格式和类型。可实现的具体的值使用下面的方法查询;

```
GLenum      readType, readFormat;
GLubyte    *pixels;
glGetIntegerv(GL_IMPLEMENTATION_COLOR_READ_TYPE, &readType);
glGetIntegerv(GL_IMPLEMENTATION_COLOR_READ_FORMAT, &readFormat);
unsigned int    bytesPerPixel = 0;
switch(readType)
{
    case GL_UNSIGNED_BYTE:
        switch(readFormat)
        {
            case GL_RGBA:
                bytesPerPixel = 4;
                break;
            case GL_RGB:
                bytesPerPixel = 3;
                break;
            case GL_LUMINANCE_ALPHA:
                bytesPerPixel = 2;
                break;
            case GL_ALPHA:
            case GL_LUMINANCE:
                bytesPerPixel = 1;
                break;
```

```
            }
            break;
        case GL_UNSIGNED_SHORT_4444:      // GL_RGBA format
        case GL_UNSIGNED_SHORT_555_1:     // GL_RGBA format
        case GL_UNSIGNED_SHORT_565:       // GL_RGB    format
            bytesPerPixel = 2;
            break;
}
pixels = (GLubyte*) malloc(width * height * bytesPerPixel);
glReadPixels(0, 0, windowWidth, windowHeight, readFormat,
                    readType, pixels );
```

你能从当前绑定的帧缓冲区读回像素值，无论它是被操作系统分配的窗口。还是一个帧缓冲区对象。因为每个缓冲区有不同的布局，你可能需要查询你想去读的缓冲区的格式。

OpenGL ES 2.0 没有拷贝一块像素数据到缓冲区的函数。替代可用的方法是创建一个贴图映射到一个像素块，使用第 9 章贴图中的贴图映射技术初始化写像素。

帧缓冲区对象

本章我们讨论帧缓冲区对象，应用程序创建它们，使用它们渲染非显示缓冲区或渲染贴图。我们以为什么需要帧缓冲区对象开始。介绍帧缓冲区对象和讨论它们是什么，一种 OpenGL ES 新的对象类型，它们和第 3 章 EGL 介绍中 EGL 窗口的不同。我们将要讨论到为什么要创建帧缓冲区对象，怎样指定和帧缓冲区对象相联系的颜色、深度、模板。然后提供一个渲染帧缓冲区对象的例子。最后我们讨论使用帧缓冲区对象的窍门和技巧，帮助我们更好的执行帧缓冲区对象。

为什么使用帧缓冲区对象

在任何应用的函数执行前，一个渲染环境和绘图窗口需要被创建和激活。渲染环境和绘图窗口被操作系统的 API 接口提供，例如 EGL。第 3 章描述了怎样去创建 EGL 环境和窗口以及怎样联系他们到渲染线程。渲染环境容纳着正确操作所要求的合适的状态。渲染窗口由操作系统提供显示在屏幕上的外观，由操作系统提供的帧缓冲区或者不显示出来的窗口，或者是 pbuffer。调用 EGL 绘制的窗口让你以像素为单位制定窗口的宽和高，无论窗口是否使用颜色、模板和深度缓冲区。

默认情况下，OpenGL ES 使用视窗系统提供的帧缓冲区绘制窗口。如果应用仅仅绘制显示的窗口，操作系统提供的帧缓冲区往往是足够的。然而，很多应用需要渲染一个贴图，使用操作系统提供的窗口经常不是好的选择。例如使用渲染贴图做动态反射和环境映射，使用多种技术例如深度视野、运动模糊、运动过进影响。

这有两个技术应用能够使用渲染一个贴图。

贴图渲染到操作系统提供的帧缓冲区，拷贝帧缓冲区合适的区域到贴图。这使用 glCopyTexImage2D 和 glCopyTexSubImage2D 执行。就像它们的名字，这两个 API 提供从帧缓冲区到贴图缓冲区的拷贝，这些操作经常是可逆向执行的。另外，唯一要求是贴图尺寸应该小于或等于帧缓冲区尺寸。

使用 pbuffer 关联贴图。我们知道操作系统提供的窗口必须联系渲染环境。对一些应用来说要求每个 pbuffer 和窗口有独立的环境是没效率的。有时操作系统要求应用程序完成先前的渲染再进行交换。这需要在管道中引入费时的 bubble（时间间隙吧）。在如此一个系统里，我们推荐不使用 bubble 缓冲区渲染贴图，因为关联环境和操作系统提供交换的开销太大。

两种方法都是渲染贴图和其它非显示画面的注意。这需要允许应用程序直接使用 OpenGL ES 的 API 渲染一个贴图和创建非显示窗口，以及使用它渲染目标。帧缓冲区对象和渲染缓冲区对象允许应用不创建另外的渲染环境正确的完成以上任务。我们不必在担心环境的搭建，使用操作系统提供的交换缓冲区功能引起时间消耗。帧缓冲区对象提供一个更好的更有效的渲染环境和非显示窗口的方法。

帧缓冲区对象 API 支持下面的操作：

只使用 OpenGL ES 函数创建帧缓冲区对象。

使用单一的 EGL 环境创建使用多个帧缓冲区对象，也就是多个帧缓冲区对象要求一个渲染环境。

创建不显示的颜色、深度、模板渲染缓冲区和贴图，把它们关联到帧缓冲区对象上。

通过多种缓冲区共享颜色、深度、模板缓冲区内容。

把贴图作为颜色、深度直接关联到帧缓冲区，避免使用拷贝操作。

帧缓冲区和渲染缓冲区对象

我们已经描述了渲染缓冲区和帧缓冲区对象，以及它们在使用渲染缓冲区代替贴图时，和操作系统提供的绘制窗口的区别。

渲染缓冲区是应用程序分配的 2D 图像缓冲区。渲染缓冲区被用来存储颜色、深度或模

板值，能作为颜色、深度或模板关联到一个帧缓冲区对象。渲染缓冲区和操作系统提供的不显示的绘制窗口 pbuffer 有很多相似的地方。渲染缓冲区不能直接被 GL 贴图使用。

帧缓冲区（经常被作为 FB O 提到）关联点的颜色、深度和模板缓冲区，描述联系到 FBO 的合适的颜色、深度和目标缓冲区的大小格式等，贴图和渲染缓冲区对象的名字被关联到 FBO。各种 2D 图像能被关联到帧缓冲区对象的颜色点上。包括存储颜色的帧缓冲区对象、中等质量的 2D 贴图或立方体面，甚至是中等质量的 3D 贴图的一个 2D 切片。同样的，各种包含着深度值的 2D 图像能被关联到一个 FBO 的深度点上。那些包括渲染缓冲区、中等质量的 2D 贴图或者存储深度值的立方体面。仅仅 2D 图像能关联一个 FBO 的模板点，FBO 是存储模板值的渲染缓冲区对象。

图 12-1 显示了帧缓冲区对象、渲染缓冲区对象和贴图的关系。注意仅仅一个颜色、深度和模板管理到一个帧缓冲区对象。

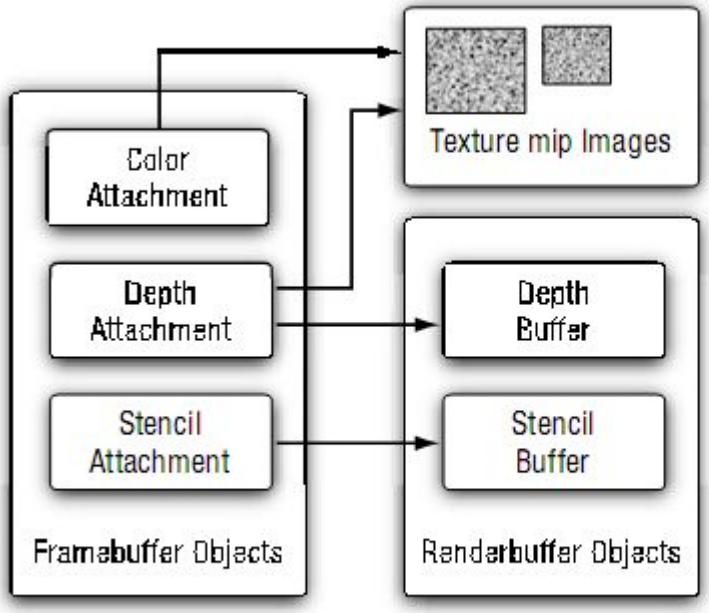


Figure 12-1 Framebuffer Objects, Renderbuffer Objects, and Textures

注意：可选的 GL_OES_texture_3D 扩展允许 3D 贴图的 2D 切片使用帧缓冲区关联。

可选的 GL_OES_depth_texture 扩展允许 2D 和立方体深度贴图

GL_OES_packed_depth_stencil 扩展允许应用使用包装的深度和模板贴图作为深度和模板缓冲区关联。

选择也个对应贴图的渲染缓冲区做帧缓冲区关联

使用容器去渲染贴图，关联贴图对象到一个帧缓冲区对象。例子包括渲染颜色缓冲区将作为一个颜色贴图使用，或渲染深度缓冲区作为深度贴图或阴影使用。

有几个原因使用渲染缓冲区替代贴图，包括：

某些图像格式不支持贴图，像深度索引值，必须使用渲染缓冲区替代。

如果不使用图像做贴图，而是使用渲染缓冲区将有这些好处。这是因为编译工具能够使用更有效率的格式存储渲染缓冲区内容，比贴图更适合渲染。编译工具仅仅要做的，知道图像不作为贴图使用。

帧缓冲区对象对比 EGL 窗口

关于 FBO 和操作系统提供的窗口不同有：

像素所有权测试，测试决定在帧缓冲区的（x_w, y_w）位置的像素是否被 OpenGL ES 当前

拥有。测试允许窗口系统控制帧缓冲区中的像素属于当前 OpenGL ES 环境。例如一个被渲染的窗口是否被 OpenGL ES 遮挡。为应用创建帧缓冲区目标，帧缓冲区测试成功，帧缓冲区对象拥有所有的像素。

窗口系统可能仅仅支持双缓冲区窗口。另一方面，帧缓冲区对象仅仅支持单缓冲区关联。

帧缓冲区对象的模板和深度缓冲区分享可能使用帧缓冲区对象，但经常不是窗口系统提供的帧缓冲区。模板和深度缓冲区和它们相应的状态经常暗指窗口系统提供的绘制窗口，因此不能被窗口共享。伴随着应用创建的帧缓冲区对象，模板和深度缓冲区被独立的创建，然后关联到帧缓冲区对象，如果需要，通过关联那些缓冲区到多帧缓冲区对象。

系统窗口提供的帧缓冲区可能支持多重采样。能够使用系统的 API 指定多重采样缓冲区。在 EGL 中，通过设定 EGL 配置的 EGL_SAMPLE_BUFFERS 指定多重采样缓冲区，应用程序创建的帧缓冲区对象不支持多重采样缓冲区关联。

创建帧缓冲区对象和渲染缓冲区对象

OpenGL ES 2.0 中创建帧缓冲区对象和渲染缓冲区对象和创建贴图和顶点缓冲区对象是相似的。

glGenRenderbuffers 函数分配渲染缓冲区对象名字。

void glGenRenderbuffers(GLsizei n, GLuint *renderbuffers)

n 返回的渲染缓冲区对象名字数目

renderbuffers 包含 n 个入口地址的矩阵的指针，矩阵包含着分配的渲染缓冲区对象

glGenRenderbuffers 分配 n 个渲染缓冲区对象名字，在 renderbuffers 中返回。glGenRenderbuffer 返回的名字是大于 0 的无符号整型数。返回的渲染缓冲区名字被标记为已使用但没有设定任何状态。0 被 OpenGL ES 保留不分配给任何渲染缓冲区。任何查询或者修改 0 渲染缓冲区将产生一个错误。

glGenFramebuffers 用于分配帧缓冲区对象。

void glGenFramebuffers(GLsizei n, GLuint *ids)

n 返回的帧缓冲区对象名字数目

ids 包含 n 个入口地址的矩阵的指针，矩阵包含着分配的帧缓冲区对象

glGenFramebuffers 分配 n 个帧缓冲区对象名字，在 ids 中返回。返回的名字是大于 0 的无符号整型数。返回的帧缓冲区名字被标记为已使用但没有设定任何状态。0 被 OpenGL ES 保留，指示操作系统提供的帧冲区。任何查询或者修改 0 帧缓冲区将产生一个错误。

使用渲染缓冲区对象

本节我们描述怎样指示数据存储、格式和渲染缓冲区图像的大小。去指定一个特殊的渲染缓冲区图像信息，需要设定缓冲区对象为当前缓冲区对象。使用 glBindRenderbuffer 函数设定当前渲染缓冲区对象。

void glBindRenderbuffer(GLenum target, GLuint renderbuffer)

target 必须被设置为 GL_RENDERBUFFER

renderbuffer 渲染缓冲区对象的名字

注意在使用 glBindRenderbuffer 绑定缓冲区前，glGenRenderbuffers 不要求去分配渲染缓冲区对象名字。虽然调用 glGenRenderbuffers 是一个好的习惯，那仍然有很多应用程序在编译时为它们的缓冲区指定常量。应用程序能指定一个不使用的渲染缓冲区对象名字去执行 glBindRenderbuffer。我们建议使用 glGenRenderbuffers 返回的名字替代应用程序自己指定缓冲区名字。

第一次调用 glBindRenderbuffer 绑定渲染缓冲区名字，如果成功分配，渲染缓冲区被分配合适的默认状态。分配的东西成为新绑定的渲染缓冲区对象。

渲染缓冲区对象的默认状态和默认值有：

以像素为单位的宽和高—默认值是 0。
内部格式—描述存储在渲染缓冲区的像素格式。必须是可渲染的颜色、深度或模板格式
颜色位数—仅仅内部格式是可渲染的颜色时有效，默认值为 0。
深度位数—仅仅内部格式是可渲染的深度时有效，默认值为 0。
模板位数—仅仅内部格式是可渲染的模板时有效，默认值为 0。

glBindRenderbuffer 能被使用去绑定一个存在的渲染缓冲区对象（一个对象被指定和使用前有合法的状态）。绑定命令不改变新的渲染缓冲区对象的状态。

一旦渲染缓冲区对象被绑定，我们能指定渲染缓冲区内的存储的图像的尺寸和格式。glRenderbufferStorage 函数指定这些信息。

```
void    glRenderbufferStorage(GLenum target,
                              GLenum internalformat,
                              GLsizei width, GLsizei height)
```

target 必须设定为 GL_RENDERBUFFER

internalformat 必须是颜色缓冲区、深度缓冲区或模板缓冲区可用的格式。

下面的格式必须支持:
GL_RGB565
GL_RGBA4
GL_RGB5_A1
GL_DEPTH_COMPONENT16
GL_STENCIL_INDEX8
下面的格式可选支持:
GL_RGB8_OES
GL_RGBA8_OES
GL_DEPTH_COMPONENT24_OES
GL_DEPTH_COMPONENT32_OES
GL_STENCIL_INDEX1_OES
GL_STENCIL_INDEX4_OES
GL_DEPTH24_STENCIL8_OES

width 渲染缓冲区以像素为单位的高度;
必须 <= GL_MAX_RENDERBUFFER_SIZE.

height 渲染缓冲区以像素为单位的高度;
必须 <= GL_MAX_RENDERBUFFER_SIZE.

glRenderbufferStorage 和 glTexImage2D 非常类似，除了没有图像数据支持。宽和高单位是像素，值大小必须被编译工具支持。最小值是 1，实际最大值使用下面的方法查询：

```
GLint maxRenderbufferSize;
glGetIntegerv(GL_MAX_RENDERBUFFER_SIZE, &maxRenderbufferSize);
```

internalformat 指出应用程序存储像素的格式
如是 GL_RGB565,GL_RGBA4, GL_RGB5_A1, GL_RGB8_OES,或 GL_RGBA8_OES，表明是颜色缓冲区。
如果是 GL_DEPTH_COMPONENT16,GL_DEPTH_COMPONENT24_OES, 或 GL_DEPTH_COMPONENT32_OES，表明是深度缓冲区。
如果是 GL_STENCIL_INDEX8, GL_STENCIL_INDEX4_OES, 或 GL_STENCIL_INDEX1_OES，表明是模板缓冲区。

渲染缓冲区对象能够作为颜色、深度、模板关联到帧缓冲区对象，不使用渲染缓冲区指

定的存储格式和尺寸。渲染缓冲区的存储格式和尺寸能被指定在渲染缓冲区被关联到帧缓冲区对象前或者后。但无论如何，在使用去渲染前应该正确指定。
注意：如果支持 GL_OES_rgb8_rgba8 扩展，GL_RGB8_OES 和 GL_RGBA8_OES 格式能被使用。
如果支持 GL_OES_depth24 或 GL_OES_depth32 扩展，GL_DEPTH_COMPONENT24_OES 或 GL_DEPTH_COMPONENT32_OES 格式能被使用。
如果支持 GL_OES_stencil1 或 GL_OES_stencil4 扩展，GL_STENCIL_INDEX1_OES 或 GL_STENCIL_INDEX4_OES 格式能被使用。
如果支持 GL_OES_packed_depth_stencil 扩展，GL_DEPTH24_STENCIL8_OES 格式能被使用。
使用帧缓冲区对象

我们描述怎样使用帧缓冲区对象去渲染一个不可见的屏幕缓冲区（例如渲染缓冲区）或者渲染一个图像。使用帧缓冲区对象和指定它的关联前，我们需要使能当前帧缓冲区对象，glBindFramebuffer 函数设定当前帧缓冲区对象。

void glBindFramebuffer(GLenum target, GLuint framebuffer)

target 必须设定为 GL_FRAMEBUFFER

framebuffer 帧缓冲区对象名字

注意 glGenFramebuffers 在使用 glBindFramebuffer 绑定前，不要求分配帧缓冲区对象名字。应用程序可以知道一个没使用的缓冲区对象名字去 glBindFramebuffer。但我们推荐应用程序调用 glGenFramebuffers 产生帧缓冲区对象名字，替代指定自己的缓冲区对象名。

第一次帧缓冲区对象被 glGenFramebuffers 绑定时，如果分配成功，帧缓冲区对象被分配合适的默认状态，分配的东西被绑定作为当前帧缓冲区对象渲染环境。
下面的状态被分配给帧缓冲区对象。
颜色附着点—为颜色缓冲区
深度附着点—为深度缓冲区
模板附着点—为模板缓冲区
帧缓冲区完整的状态—帧缓冲区是否在完整的状态，和能否被渲染。
每个附着点，下面的信息被指定：

对象类型—指定关联到附着点的对象类型。如果是渲染缓冲区被关联，类型是 GL_RENDERBUFFER。如果贴图对象被关联，类型是 GL_TEXTURE，默认值是 GL_NONE。

对象名字—指定对象附着的名字。可能是渲染缓冲区对象名或贴图对象名，默认值是 0。

贴图等级—如果贴图对象被附着，这指定贴图的中等级别被关联到附着点，默认值是 0。

立方体贴图面—如果贴图对象被附着，而且贴图对象是立方体贴图，指定立方体贴图的那个面被关联到附着点，默认值是 GL_TEXTURE_CUBE_MAP_POSITIVE_X。

贴图 Z 偏置—如果 GL_OES_texture_3D 扩展被支持，指示 3D 贴图的哪个 2D 切片被使用做附着点。默认值是 0。

glBindFramebuffer 能被使用绑定一个存在的帧缓冲区对象（前面已分配和使用因此有有效状态的对象）不改变新的绑定帧缓冲区对象的状态。

一旦一个帧缓冲区对象被绑定，颜色、深度和模板为当前对象，将被设定为渲染缓冲区对象或贴图。图 12-1 显示颜色附着能设定渲染缓冲区存储颜色数据，或者一个中等级别的 2D 贴图或者一个立方体贴图的一个面，或者是 3D 贴图的一个中等级别的 2D 切片。深度附着能被设定渲染缓冲区存储深度值，或者中等级别的 2D 贴图深度值，或者立方体贴图一个面的深度值。模板附着必须被设定渲染缓冲区存储模板值。
关联渲染缓冲区成为帧缓冲区附着

glFramebufferRenderbuffer 被使用关联渲染缓冲区成为帧缓冲区附着点。

```
void    glFramebufferRenderbuffer(GLenum target,
                                  GLenum attachment,
                                  GLenum renderbuffertarget,
                                  GLuint renderbuffer)
```

target 必须设定为 GL_FRAMEBUFFER

attachment 附着的必须是下面枚举值之一:

GL_COLOR_ATTACHMENT0

GL_DEPTH_ATTACHMENT

GL_STENCIL_ATTACHMENT

renderbuffertarget 必须设定为 GL_RENDERBUFFER

renderbuffer 将被使用的附着的渲染缓冲区对象. 渲染缓冲区必须是 0 或者存在的渲染缓冲区对象名字

如果 glFramebufferRenderbuffer 调用后 renderbuffer 不是 0，渲染缓冲区对象被当作由参数 attachment 决定的、新的颜色、深度或模板附着点。

附着点状态被修改为：

对象类型为 GL_RENDERBUFFER。

对象名字为 renderbuffer

贴图等级、立方体贴图面、贴图 Z 偏置为 0。

新附着的渲染缓冲区对象状态或内容不改变。

如果 glFramebufferRenderbuffer 调用后 renderbuffer 是 0，由参数 attachment 决定的、新的颜色、深度或模板附着点被分离，复位到 0。

关联 2D 贴图成为帧缓冲区附着

glFramebufferTexture2D 函数被用于附着中等质量的 2D 贴图或立方体贴图的一个面到帧缓冲区附着点上。它能使用关联贴图为颜色或深度附着。不能用于模板贴图附着。

```
void    glFramebufferTexture2D(GLenum target,
                               GLenum attachment,
                               GLenum textarget,
                               GLuint texture,
                               GLint  level)
```

target 必须设定为 GL_FRAMEBUFFER

attachment 附着的必须是下面枚举值之一:

GL_COLOR_ATTACHMENT0

GL_DEPTH_ATTACHMENT

textarget 指定贴图目标. 这个值指定 glTexImage2D 的 textarget 参数。

texture 指定贴图对象

level 指定贴图对象的质量为中等

如果调用 glFramebufferTexture2D 的输入参数 texture 不是 0，那么颜色或深度附着设定到贴图，其它情况下，glFramebufferTexture2D 产生错误，不改变帧缓冲区状态。

附着后状态被修改为：

对象类型为 GL_TEXTURE

对象名字为 texture

贴图等级为 level

立方体贴图的面为下面中的一个

GL_TEXTURE_CUBE_MAP_POSITIVE_X
GL_TEXTURE_CUBE_MAP_POSITIVE_Y
GL_TEXTURE_CUBE_MAP_POSITIVE_Z
GL_TEXTURE_CUBE_MAP_NEGATIVE_X
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z
贴图 Z 偏置为 0

新的贴图对象的状态和内容不会被 glFramebufferTexture2D 修改。注意贴图对象的状态和图像关联到帧缓冲区对象后能被修改。

如果使用 texture 为 0 调用 glFramebufferTexture2D，颜色和深度附着被分离，复位到 0。关联一个 3D 贴图图像作为帧缓冲区附着

glFramebufferTexture3DOES 被用于关联一个指定的 3D 贴图的中等质量的 2D 切片到帧缓冲区附着点。这仅仅用于关联一个贴图作为颜色附着。一个 3D 贴图深度和模板附着不能使用。看第 9 章贴图，对 3D 贴图的详细描述。

```
void    glFramebufferTexture3DOES(GLenum target,
                                  GLenum attachment,
                                  GLenum textarget,
                                  GLuint texture,
                                  GLint  level,
                                  GLint  zoffset)
```

target	必须设定为 GL_FRAMEBUFFER
attachment	附着的必须是下面枚举值之一： GL_COLOR_ATTACHMENT0
textarget	指定贴图目标，这个值是 glTexImage3DOES 的 textarget 参数指定的。
texture	指定贴图对象
level	指定贴图图像为中等质量水平
zoffset	指定 z-offset，用于识别 3D 贴图的 2D 切片，必须是 0 到 (深度值 - 1),深度被 glTexImage3DOES 指定。

如果使用 texture 不为 0 调用 glFramebufferTexture3DOES，颜色缓冲区附着设定为 texture，别的情况下 glFramebufferTexture3DOES 产生一个错误，不改变帧缓冲区状态设定。附着点的状态将被修改成：
对象类型为 GL_TEXTURE
对象名字为 texture
贴图等级为 level
贴图立方体面为 0
贴图 Z 偏置为 zoffset

新的附着贴图对象状态和内容不能使用 glFramebufferTexture3DOES 修改，但贴图对象的状态和图像在附着到帧缓冲区对象后能被修改。

如果使用 texture 等于 0 调用 glFramebufferTexture3DOES，那么颜色附着被分离复位到 0。

一个有趣的问题出现：如果我们渲染一个贴图时，同时使用这个贴图对象作为片段着色器的贴图，将发生什么事情。OpenGL ES 编译器是否会产生一个错误？在一些情况下 OpenGL ES 编译器决定是否一个贴图对象被用于贴图输入和帧缓冲区附着我们当前的绘制。glDrawArrays 和 glDrawElements 将产生一个错误。为了确定 glDrawArrays 和

glDrawElements 能尽可能快的执行，需要进行一些检查。代替产生错误，渲染结果是不定的。这就需要应用程序确保这种情况不会发生。

检查帧缓冲区完整性

帧缓冲区对象被用于渲染目标前，需要完整定义。如果当前使用的帧缓冲区对象不是完整的，OpenGL ES 函数绘制基元和读像素操作将失败，产生一个错误说明原因为帧缓冲区是不完整的。

帧缓冲区完整包括:

确保颜色、深度、模板缓冲区附着是合法的。颜色附着如果是 0 也是合法的（也就是说，没有附着）或者是颜色渲染缓冲区对象、贴图对象。对颜色渲染可用的格式有：GL_RGB565, GL_RGBA4,GL_RGB5_A1, 和 optionally GL_RGB8_OES 及 GL_RGBA8_OES（扩展被支持）。深度附着是合法的，如果值是 0 或者深度渲染缓冲区对象或者贴图深度值。合法的深度渲染缓冲区格式有 GL_DEPTH_COMPONENT16 和 GL_DEPTH_COMPONENT24_OES 及 GL_DEPTH_COMPONENT32_OES（扩展被支持）。模板附着是合法的，如果值是 0 或者模板渲染缓冲区对象或者贴图深度值。合法的模板渲染缓冲区格式有 GL_STENCIL_INDEX8 和 GL_STENCIL_INDEX1_OES 及 GL_STENCIL_INDEX4_OES（扩展被支持）。

最小一个有效附着被支持。如果没有附着任何东西，就没有绘制任何物体或读任何内容，帧缓冲区是不完整。

有效的附着帧缓冲区对象的附着必须有相同的宽和高。

渲染目标的颜色、深度和模板内部的组成格式经过特殊的编译能被作为最终的窗口的结果。虽然有多种颜色、深度和模板内部的组成格式，但不是所有的格式被编译器支持。例如编译器可能不支持 16 位的深度和 8 位的模板缓冲区渲染。这种情况下，帧缓冲区对象使用 GL_DEPTH_COMPONENT16 和 GL_STENCIL_INDEX8 将不支持。

glCheckFramebufferStatus 函数被使用核实一个帧缓冲区对象是否完整。

GLenum glCheckFramebufferStatus(GLenum target)

target 必须设定为 GL_FRAMEBUFFER

如果目标不等于 GL_FRAMEBUFFER，glCheckFramebufferStatus 返回 0，如果是 GL_FRAMEBUFFER，返回下面枚举值中的一个：

GL_FRAMEBUFFER_COMPLETE—帧缓冲区是完整的

GL_FRAMEBUFFER_INCOMPLETE_ATTACHMENT—帧缓冲区是不完整的，可能的原因是附着是 0 或不是一个有效的贴图或渲染缓冲区对象

GL_FRAMEBUFFER_MISSING_ATTACHMENT—帧缓冲区里没有有效的附着

GL_FRAMEBUFFER_INCOMPLETE_DIMENSIONS—附着没有相同的宽和高

GL_FRAMEBUFFER_INCOMPLETE_FORMATS—附着的内部格式不能渲染

GL_FRAMEBUFFER_UNSUPPORTED—附着在帧缓冲区的内部格式导致目标不能渲染

如果当前绑定的帧缓冲区对象不完整，使用帧缓冲区对象读写像素会失败。这意味着基元绘制 glDrawArrays 和 glDrawElements，读帧缓冲区 glReadPixels ，glCopyTexImage2D, glCopyTexSubImage2D, 和 glCopyTexSubImage3DOES 会产生 GL_INVALID_FRAMEBUFFER_OPERATION 错误。

删除帧缓冲区对象和渲染缓冲区对象

使用完渲染缓冲区对象后，它们能够被删除，删除渲染缓冲区和帧缓冲区是非常相似删除贴图对象。

渲染缓冲区对象删除使用 glDeleteRenderbuffers 函数。

void glDeleteRenderbuffers(GLsizei n,

GLuint *renderbuffers)

n 需要删除的渲染缓冲区对象名字

renderbuffers 需要删除的 n 个渲染缓冲区对象名字的矩阵的指针

glDeleteRenderbuffers 删除 renderbuffers 指定的渲染缓冲区对象。一旦渲染缓冲区对象被删除，它没有状态关联，被标志为未使用，能够被作为新的渲染缓冲区对象使用。当删除一个当前绑定的渲染缓冲区对象时，渲染缓冲区对象被删除，当前渲染缓冲区绑定被回复成 0。如果删除的渲染缓冲区对象是非法的或是 0，它们被忽略（不会产生错误）。如果渲染缓冲区对象附着到当前绑定的帧缓冲区对象，将先分离附着然后删除。

使用 glDeleteFramebuffers 删除帧缓冲区对象。

void glDeleteFramebuffers(GLsizei n, GLuint *framebuffers)

n 需要删除的帧缓冲区对象名字

framebuffers 需要删除的帧缓冲区对象名字

glDeleteFramebuffers 删除 frame-buffers。指定的渲染缓冲区对象。一旦帧缓冲区对象被删除，它没有状态关联，被标志为未使用，能够被作为新的帧缓冲区对象使用。当删除一个当前绑定的帧缓冲区对象时，帧缓冲区对象被删除，当前帧缓冲区绑定被回复成 0。如果删除的帧缓冲区对象是非法的或是 0，它们被忽略（不会产生错误）。

删除作为帧缓冲区附着使用的渲染缓冲区对象

如果删除作为帧缓冲区附着使用的渲染缓冲区对象，会发生什么事情？如果删除作为当前帧缓冲区附着使用的渲染缓冲区对象，glDeleteRenderbuffers 将复位附着为 0。如果删除作为不是当前帧缓冲区附着使用的渲染缓冲区对象，glDeleteRenderbuffers 将不复位附着为 0。这是应用程序需要去完成的工作。

读像素和帧缓冲区对象

glReadPixels 函数从颜色缓冲区里读像素到开发者分配的缓冲区里。被读的颜色缓冲区可能是操作系统提供的帧缓冲区分配的颜色缓冲区或者当前绑定的帧缓冲区对象附着的颜色缓冲区。glReadPixels 支持两种组成格式和类型：GL_RGB 和 GL_UNSIGNED_BYTE 或者是应用程序使用 GL_IMPLEMENTATION_COLOR_READ_FORMAT 和 GL_IMPLEMENTATION_COLOR_READ_TYPE 查询返回的格式和类型。应用程序查询返回的格式和类型是当前附着颜色缓冲区的格式和类型。它的值会随着当前绑定的帧缓冲区改变而变化。必须使用查询去决定输入到 glReadPixels 的值。

例子

让我们看一些使用帧缓冲区的例子，例 12-1 显示怎样使用帧缓冲区对象渲染贴图。例子中使用帧缓冲区对象渲染一个贴图，然后使用贴图绘制一个操作系统提供的帧缓冲区正方形（屏幕）。图 12-2 显示这个图像。

例 12-1 渲染贴图

```
GLuint     framebuffer;
GLuint     depthRenderbuffer;
GLuint     texture;
GLint      texWidth = 256, texHeight = 256;
GLint      maxRenderbufferSize;
glGetIntegerv(GL_MAX_RENDERBUFFER_SIZE, &maxRenderbufferSize);
// check if GL_MAX_RENDERBUFFER_SIZE is >= texWidth and texHeight
if((maxRenderbufferSize <= texWidth) ||
   (maxRenderbufferSize <= texHeight))
{
```

```
    // cannot use framebuffer objects as we need to create
    // a depth buffer as a renderbuffer object
    // return with appropriate error
}
// generate the framebuffer, renderbuffer, and texture object names
glGenFramebuffers(1, &framebuffer);
glGenRenderbuffers(1, &depthRenderbuffer);
glGenTextures(1, &texture);
// bind texture and load the texture mip-level 0
// texels are RGB565
// no texels need to be specified as we are going to draw into
// the texture
glBindTexture(GL_TEXTURE_2D, texture);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texWidth, texHeight,
                    0, GL_RGB, GL_UNSIGNED_SHORT_5_6_5, NULL);
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);
// bind renderbuffer and create a 16-bit depth buffer
// width and height of renderbuffer = width and height of
// the texture
glBindRenderbuffer(GL_RENDERBUFFER, depthRenderbuffer);
glRenderbufferStorage(GL_RENDERBUFFER, GL_DEPTH_COMPONENT16,
                                texWidth, texHeight);
// bind the framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
// specify texture as color attachment
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                                 GL_TEXTURE_2D, texture, 0);
// specify depth_renderbufer as depth attachment
glFramebufferRenderbuffer(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                                    GL_RENDERBUFFER, depthRenderbuffer);
// check for framebuffer complete
status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if(status == GL_FRAMEBUFFER_COMPLETE)
{
    // render to texture using FBO
    // clear color and depth buffer
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // load uniforms for vertex and fragment shader
    // used to render to FBO. The vertex shader is the
    // ES 1.1 vertex shader described as Example 8-8 in
```

```
    // Chapter 8. The fragment shader outputs the color
    // computed by vertex shader as fragment color and
    // is described as Example 1-2 in Chapter 1.
    set_fbo_texture_shader_and_uniforms();
    // drawing commands to the framebuffer object
    draw_teapot();
    // render to window system provided framebuffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    // Use texture to draw to window system provided framebuffer
    // We draw a quad that is the size of the viewport.
    //
    // The vertex shader outputs the vertex position and texture
    // coordinates passed as inputs.
    //
    // The fragment shader uses the texture coordinate to sample
    // the texture and uses this as the per-fragment color value.
    set_screen_shader_and_uniforms();
    draw_screen_quad();
}
// cleanup
glDeleteRenderbuffers(1, &depthRenderbuffer);
glDeleteFramebuffers(1, &framebuffer);
glDeleteTextures(1, &texture);
```

Figure 12-2 Render to Color Texture

例 12-1，我们创建了帧缓冲区、贴图和使用合适的 glGen***函数创建深度渲染缓冲区对象。帧缓冲区使用颜色附着一个贴图对象(texture)，深度附着渲染缓冲区对象(depthRenderbuffer)。

创建这些东西前，我们查询最大的渲染缓冲区尺寸(GL_MAX_RENDERBUFFER_SIZE)，确定编译器支持的最大渲染缓冲区尺寸小于等于作为颜色附着的贴图的宽和高。这确保我们成功的创建深度渲染缓冲区，使在帧缓冲区上用时的附着。

对象被创建后，我们调用 glBindTexture(texture)确定当前贴图绑定到贴图对象。使用 glTexImage2D 指定中等等级的贴图。注意输入参数 pixels 是 NULL。这是因为我们渲染的是整个贴图区域，因此不需要知道任何输入数据做覆盖区。

使用 glBindRenderbuffer 和绑定 depthRenderbuffer 对象，调用 glRenderbufferStorage 分配 16 位深度缓冲区的存储空间。

使用 glBindFramebuffer 绑定帧缓冲区对象，texture 作为颜色附着到 framebuffer，depthRenderbuffer 作为深度附着到 framebuffer。

在开始渲染帧缓冲区前，需要检查帧缓冲区是否完整。一旦帧缓冲区完整，我们通过调用 glBindFramebuffer(GL_FRAMEBUFFER, 0)复位当前绑定帧缓冲区大操作系统提供的帧缓冲区。我们能使用在 framebuffer 里的 texture 做渲染目标，绘制操作系统提供的帧缓冲区。

例 12-1，深度缓冲区被附着到帧缓冲区渲染目标。例 12-2 我们看到怎样使用时的贴图作为深度缓冲区附着到帧缓冲区。如果扩展 OES_depth_texture 被支持，这个特性可以使用。应用程序可以使用帧缓冲区附着光源渲染深度贴图。渲染的深度缓冲区能被用于阴影映射计算每个频道在阴影上的百分比。图 12-3 显示了产生的图像。

例 12-2 渲染深度贴图

```
#define COLOR_TEXTURE     0
#define DEPTH_TEXTURE     1
GLuint      framebuffer;
GLuint      textures[2];
GLint       texWidth = 256, texHeight = 256;
// generate the framebuffer, & texture object names
glGenFramebuffers(1, &framebuffer);
glGenTextures(2, textures);
// bind    color texture and load the texture mip-level 0
// texels are RGB565
// no texels need to specified as we are going to draw into
// the texture
glBindTexture(GL_TEXTURE_2D, textures[COLOR_TEXTURE]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, texWidth, texHeight,
                    0, GL_RGB, GL_UNSIGNED_SHORT_5_6_5, NULL);
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);
// bind    depth texture and load the texture mip-level 0
// no texels need to specified as we are going to draw into
// the texture
glBindTexture(GL_TEXTURE_2D, textures[DEPTH_TEXTURE]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, texWidth,
                    texHeight, 0, GL_DEPTH_COMPONENT, GL_UNSIGNED_SHORT,
```

```
                NULL);
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_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
// bind the framebuffer
glBindFramebuffer(GL_FRAMEBUFFER, framebuffer);
// specify texture as color attachment
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                       GL_TEXTURE_2D, textures[COLOR_TEXTURE], 0);
// specify texture as depth attachment
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, textures[DEPTH_TEXTURE], 0);
// check for framebuffer complete
status = glCheckFramebufferStatus(GL_FRAMEBUFFER);
if (status == GL_FRAMEBUFFER_COMPLETE)
{
    // render to color & depth textures using FBO
    // clear color & depth buffer
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // load uniforms for vertex and fragment shader
    // used to render to FBO. The vertex shader is the
    // ES 1.1 vertex shader described as Example 8-8 in
    // Chapter 8. The fragment shader outputs the color
    // computed by vertex shader as fragment color and
    // is described as Example 1-2 in Chapter 1.
    set_fbo_texture_shader_and_uniforms();
    // drawing commands to the framebuffer object
    draw_teapot();
    // render to window system provided framebuffer
    glBindFramebuffer(GL_FRAMEBUFFER, 0);
    // Use depth texture to draw to window system framebuffer
    // We draw a quad that is the size of the viewport.
    //
    // The vertex shader outputs the vertex position and texture
    // coordinates passed as inputs.
    //
    // The fragment shader uses the texture coordinate to sample
    // the texture and uses this as the per-fragment color value.
    set_screen_shader_and_uniforms();
    draw_screen_quad();
}
// cleanup
```

```
glDeleteRenderbuffers(1, &depthRenderbuffer);
glDeleteFramebuffers(1, &framebuffer);
glDeleteTextures(1, &texture);
```

Figure 12-3 Render to Depth Texture

注意：不可见渲染缓冲区的宽和高不必是 2 的整次方。OpenGL ES 支持使用基本中等级别的贴图和 GL_CLAMP_TO_EDGE 模式支持非 2 整次方贴图。通过 OES_texture_npot 扩展，这些规则是宽松的。

执行窍门和技巧

我们现在讨论一下使用帧缓冲区对象必须仔细考虑的窍门。

避免频繁的交换操作系统提供的帧缓冲区渲染和帧缓冲区对象渲染。这是因为 OpenGL ES 2.0 使用很多编译工具使用小区块渲染技术。这个技术使用帧缓冲区的一小块（也就是区域）内部的内存存储颜色、深度和模板值。内部的区域使用整次方是非常有效的。对比外部内存有更好的延迟和带宽。渲染完一个片段后，片段被写到设备（或系统）内存。每次你交换一个目标到另一个，附着的贴图和渲染缓冲区将需要重新渲染、保持、修复。这非常费时。最好的方法是渲染一个第一个场景的合适的帧缓冲区，然后使用 eglSwapBuffers 函数交换操作系统提供的帧缓冲区。

不要每帧创建和销毁帧缓冲区和渲染缓冲区目标（或者其他像这一样的大量的数据对象）

尽量避免修改贴图作为渲染目标附着到帧缓冲区对象（使用 glTexImage2D,glTexSubImage2D,glCopyTexImage2D 等等）。

设定 glTexImage2D 和 glTexImage3DOES 的参数 pixels 为 NULL。如果整个贴图图像被作为原始数据渲染将不能在任何地方使用。如果你希望图像有定义的像素值，确定在绘制贴图前，你已使用 glClear 清除贴图图像。

分享深度和模板渲染缓冲区做帧缓冲区对象的附着，尽可能保持内存消耗最小。意识到我们的缓冲区有有限的宽和高是相同的。在 OpenGL ES 的未来版本中各种帧缓冲区附着规则必须可能是宽松的，容易分享。

OpenGL ES 2.0 先进的编程

本章我们讨论一些你已经在本书学过的OpenGL ES 2.0先进的使用方法。OpenGL ES 2.0中有很多灵活的先进渲染技术。本章我们学习下面的技术。

每个片段的光照

环境映射

粒子系统

图像后加工

投影贴图

3D 贴图使用噪声

程序贴图

每个片段的光照

第 8 章，顶点着色器我们介绍了能被应用于顶点着色器的光照方程计算每个顶点的光照。一般的为了实现更高质量的光照方程，我们基元每个片段计算光照方程。本章我们举例说明基于片段的计算散射、环境和镜面反射光。例子在 RenderMonkey 工作区的文件夹 Chapter_13/PerFragmentLighting/PerFragmentLighting.rfx 下，显示的图片看图 13-1

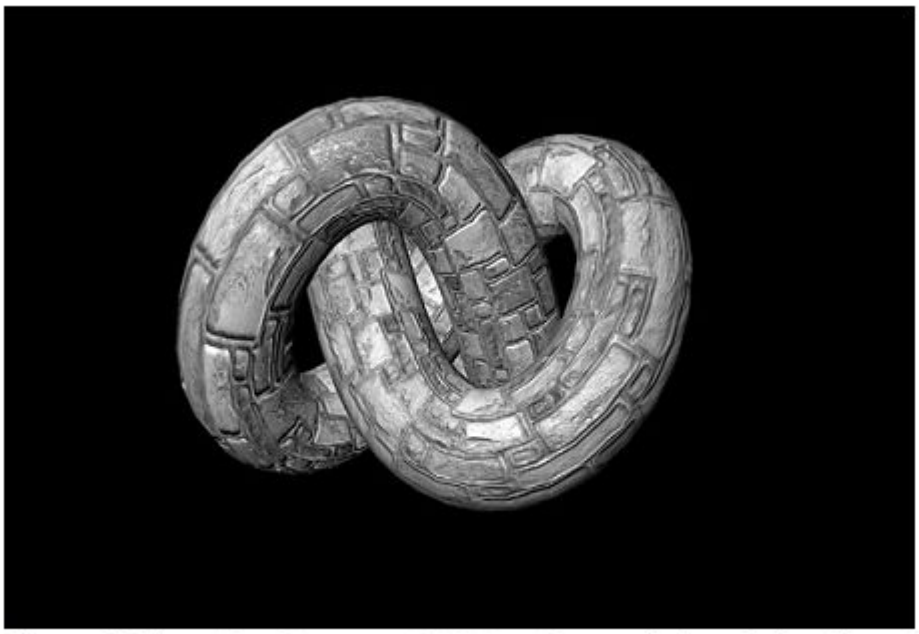


Figure 13-1 Per-Fragment Lighting Example (see Color Plate 4)

正常映射的光照

在我们进入 RenderMonkey 工作区的着色器细节前，首先讨论例子使用的方法。最简单的计算片段光照的方法是在片段着色器中使用顶点法线插补，移动光计算到片段着色器。然而对散射而言，每顶点计算光照这不是好的效果。更先进的方法是变形法线矢量，因为线性插补的原因，将产生人工痕迹，而光照质量仅仅得到有限的提高。为实现更高质量的光照，基于每个片段计算光照，使用法线匹配额存储每个纹理的法线将提供更多优良的细节。

正常的映射是一个 2D 贴图存储着每个纹理的矢量。红色通道代表 x 构成，绿色通道代表 y 构成，蓝色通道代表 z 构成。正常的映射像 GL_RGB8 使用 GL_UNSIGNED_BYTE 类型，值范围在[0, 1]。着色器里那些值需要重新定标到[-1, 1]范围。下面的片段着色器代码显示了你应该为映射做的事情。

// Fetch the tangent space normal from normal map

```
vec3 normal = texture2D(s_bumpMap, v_texcoord).xyz;

// Scale and bias from [0, 1] to [-1, 1] and normalize
normal = normalize(normal * 2.0 - 1.0);
```

就像你看到的，着色器代码将颜色值乘 2 减 1。结果在范围[0, 1]转变到[C1, 1]范围。另外如果在你的正常映射非标准化的数据，在片段着色器里需要标准化。如果你正常的映射是标准矢量，可以跳过这个步骤。

另一个处理片段光的问题是片段光必须在贴图法线存储的空间进行。为最小化片段着色器里的计算，我们我们不必转化从正常匹配取得的结果。一种实现方法是存储你正常匹配的法线。也就是说，法线匹配的法线矢量代表了外部空间的法线矢量。而光和方向矢量将从顶点着色器转变到外部空间，直接使用正常匹配的值。显着问题是存储在世界空间的法线贴图。最重要的是，该对象已被认为是静态的，因为没有转换可以发生在这个对象上。另一个重要问题是，空间上同样的表面不同方向将无法共享相同纹理在正常的匹配下，这可能会导致更大的地图。

好的解决是使用世界空间法线匹配去存储在切线空间的正常匹配。切线空间被定义为每个顶点三个轴：法线的、逆法线的和切线的。在贴图空间存储的法线被存储到切线空间。然后我们计算所有的光方程，转化我们的输入光矢量到切线空间，那些光矢量能直接被使用，在法线匹配上。切线空间典型的作为预处理进行计算。逆法线和切线被附加到顶点属性数据中。这个工作被 RenderMonkey 自动完成，它为任何有顶点法线和贴图坐标的模型计算切线空间。

着色器光照

一旦我们有切线空间的法线和切线空间的矢量，我们能计算每个片段的光照，首先让我们看例 13-1 的顶点着色器

例 13-1 顶点着色器每个频道的光照

```
uniform mat4 u_matViewInverse;
uniform mat4 u_matViewProjection
uniform vec3 u_lightPosition;
uniform vec3 u_eyePosition;
varying vec2 v_texcoord;
varying vec3 v_viewDirection;
varying vec3 v_lightDirection;
attribute vec4 a_vertex;
attribute vec2 a_texcoord0;
attribute vec3 a_normal;
attribute vec3 a_binormal;
attribute vec3 a_tangent;

void main(void)
{
    // Transform eye vector into world space
    vec3 eyePositionWorld =
                (u_matViewInverse * vec4(u_eyePosition, 1.0)).xyz;

    // Compute world space direction vector
```

```
    vec3 viewDirectionWorld = eyePositionWorld - a_vertex.xyz;

    // Transform light position into world space
    vec3 lightPositionWorld =
                (u_matViewInverse * vec4(u_lightPosition, 1.0)).xyz;

    // Compute world space light direction vector
    vec3 lightDirectionWorld = lightPositionWorld - a_vertex.xyz;

    // Create the tangent matrix
    mat3 tangentMat = mat3(a_tangent,
                           a_binormal,
                           a_normal);

    // Transform the view and light vectors into tangent space
    v_viewDirection = viewDirectionWorld * tangentMat;
    v_lightDirection = lightDirectionWorld * tangentMat;

    // Transform output position
    gl_Position = u_matViewProjection * a_vertex;
        // Pass through texture coordinate
    v_texcoord = a_texcoord0.xy;
}
```

我们有两个 uniform 矩阵需要做顶点着色器的输入 u_matViewInverse 和 u_matViewProjection，u_matViewInverse 是视图矩阵的逆矩阵。这个矩阵被用于传递光线矢量和眼睛矢量（在可视空间里）到世界空间。在主函数 main 最前面的四句声明里执行这个转换和计算世界空间的光照矢量和可视矢量。着色器的下一步是创建切线矩阵，顶点的切线空间存储了三个矢量属性：a_normal, a_binormal, 和 a_tangent。这三个矢量定义了切线空间的三个坐标轴。我们构造的 3×3 的矩阵组成切线空间 tangentMat 的三个向量。

下一步通过乘以 tangentMat 矩阵，转变视图和方向矢量为切线空间。记着我们的目标是得到切线空间标准法线匹配的视图和方向矢量空间。在顶点着色器做这些转变后，而不是在片段着色器里完成最后，计算最终的 gl_Position 里的输出位置和方向，使用 v_texcoord 传递贴图坐标到片段着色器中。

现在我们有视觉空间里的视图和方向矢量以及传递到片段着色器的贴图坐标变量。下一步片段着色器里实际的光照看例 13-2.

例 13-2 片段着色器每片段的光照

```
precision mediump float;
uniform vec4 u_ambient;
uniform vec4 u_specular;
uniform vec4 u_diffuse;
uniform float u_specularPower;
uniform sampler2D s_baseMap;
uniform sampler2D s_bumpMap;
varying vec2 v_texcoord;
```

```
varying vec3 v_viewDirection;
varying vec3 v_lightDirection;
void main(void)
{
   // Fetch basemap color
   vec4 baseColor = texture2D(s_baseMap, v_texcoord);

   // Fetch the tangent-space normal from normal map
   vec3 normal = texture2D(s_bumpMap, v_texcoord).xyz;

   // Scale and bias from [0, 1] to [-1, 1] and normalize
   normal = normalize(normal * 2.0 - 1.0);

   // Normalize the light direction and view direction
   vec3 lightDirection = normalize(v_lightDirection);
   vec3 viewDirection = normalize(v_viewDirection);

   // Compute N.L
   float nDotL = dot(normal, lightDirection);

   // Compute reflection vector
   vec3 reflection = (2.0 * normal * nDotL) - lightDirection;

   // Compute R.V
   float rDotV = max(0.0, dot(reflection, viewDirection));

   // Compute Ambient term
   vec4 ambient = u_ambient * baseColor;

   // Compute Diffuse term
   vec4 diffuse = u_diffuse * nDotL * baseColor;

   // Compute Specular term
   vec4 specular = u_specular * pow(rDotV, u_specularPower);

   // Output final color
   gl_FragColor = ambient + diffuse + specular;
}
```

片段着色器第一部分为环境、散射和镜面反射声明一系列 uniform，这些值存储在 u_ambient, u_diffuse, 和 u_specular 变量中。着色器也配置两个采样器 s_baseMap 和 s_bumpMap，他们各自绑定基本的颜色和法线映射。

片段着色器第一步从颜色映射取得基本的颜色值，从法线匹配取得基本的法线值。更详细描述是法线矢量从贴图映射定标偏移标准化到范围在[-1, 1]的单位矢量得到。下一步光线矢量和视图矢量标准化存储到 lightDirection 和 viewDirection。标准化是必需的因为

变量通过基元进行插补。插补是线性的。两个矢量线性插补结果可能是非线性。为避免这个痕迹，片段着色器里的变量必须标准化。

光方程

现在我们有在同一空间里标准化过的法线、光矢量、方向矢量。这些输入是计算光照方程必须的，让我们输入我们需要的计算光照方程。着色器里光照方程计算如下：

$$Ambient = k_{Ambient} \times C_{Base}$$

$$Diffuse = k_{Diffuse} \times N \cdot L \times C_{Base}$$

$$Specular = k_{Specular} \times pow(max(R \cdot V, 0.0), k_{specular\ Power}$$

K 是来自 u_ambient,u_diffuse, 和 u_specular 的 uniform 变量的环境、散射和镜面反射颜色的常量。C_{Base} 是取自基本贴图映射的基本颜色。光照矢量和法线矢量计算结果被存储到着色器的 nDotL 变量中。这个值用于计算散射光。最后镜面反射方程中的 R 来自方程：

$$R = 2 \times N \times (N \cdot L) - L$$

注意反射矢量也要 N•L，计算散射光的方程也能在计算镜面反射时再应用。最后计算结果存储到着色器的 ambient, diffuse,和 specular 变量中。所有的结果相加后存储到 gl_FragColor 输出变量。结果就是来自法线映射的基于每个片段光照的物体法线数据。

有很多基于每个片段的光照变种。一种普遍的技术是和使用贴图特殊的遮挡位和贴图一块存储一个特殊的指数。这允许产生镜面反射光在物体表面变异的效果。在片段着色器里使用切线空间计算光照方程是很多现代游戏的典型应用。当然这也可以增加其他的光照，更多材料属性或其它的东西。

环境映射

下一个技术—和先前的技术有联系—使用立方体贴图执行环境映射。例子看 RenderMonkey 文件夹里的 Chapter_13/Environment Mapping/EnvironmentMapping.rfx，显示结果看图 13-2

Figure 13-2 Environment Mapping Example (see Color Plate 5)

环境映射的观念是在物体上渲染环境的映射。第 9 章贴图，介绍了立方体贴图，存储环境映

射普遍使用的。在 RenderMonkey 例子里，在立方体贴图存储了一个山脉的场景。立方体贴图能使用一个场景中心的摄像机产生，使用 90 度视角拍摄每个方向和反方向。对动态的反射改变，使用帧缓冲区每帧动态的渲染这样一个立方体贴图。为静态环境这种工作能用预处理存储静态立方体贴图处理。

顶点着色器的环境映射的例子看例 13-3

例 13-3 顶点着色器环境映射

```
uniform mat4 u_matViewInverse;
uniform mat4 u_matViewProjection;
uniform vec3 u_lightPosition;
uniform vec3 u_eyePosition;
varying vec2 v_texcoord;
varying vec3 v_lightDirection;
varying vec3 v_normal;
varying vec3 v_binormal;
varying vec3 v_tangent;
attribute vec4 a_vertex;
attribute vec2 a_texcoord0;
attribute vec3 a_normal;
attribute vec3 a_binormal;
attribute vec3 a_tangent;

void main(void)
{
   // Transform light position into world space
   vec3 lightPositionWorld =
        (u_matViewInverse * vec4(u_lightPosition, 1.0)).xyz;

   // Compute world-space light direction vector
   vec3 lightDirectionWorld = lightPositionWorld - a_vertex.

   // Pass the world-space light vector to the fragment shad
   v_lightDirection = lightDirectionWorld;

   // Transform output position
   gl_Position = u_matViewProjection * a_vertex;

   // Pass through other attributes
   v_texcoord = a_texcoord0.xy;
   v_normal   = a_normal;
   v_binormal = a_binormal;
   v_tangent  = a_tangent;
}
```

本例中顶点着色器相对前面的片段着色器例子是非常简单的，不同是代替传递光方向矢量到切线空间，我们保持光矢量在世界空间。必须这样做的原因是我们最后需要使用世界空

间反射矢量获取立方体贴图。代替传递光照矢量到切线空间，我们传递切线空间的法线矢量到世界空间。这样顶点着色器传递法线、逆法线和切线变量到片段着色器，构造切线矩阵。片段着色器环境映射的例子看 13-4：

例 13-4 片段着色器环境映射

```
precision mediump float;
uniform vec4 u_ambient;
uniform vec4 u_specular;
uniform vec4 u_diffuse;
uniform float u_specularPower;
uniform sampler2D s_baseMap;
uniform sampler2D s_bumpMap;
uniform samplerCube s_envMap;
varying vec2 v_texcoord;
varying vec3 v_lightDirection;
varying vec3 v_normal;
varying vec3 v_binormal;
varying vec3 v_tangent;
void main(void)
{
   // Fetch basemap color
   vec4 baseColor = texture2D(s_baseMap, v_texcoord);

   // Fetch the tangent-space normal from normal map
   vec3 normal = texture2D(s_bumpMap, v_texcoord).xyz;

   // Scale and bias from [0, 1] to [-1, 1]
   normal = normal * 2.0 - 1.0;

   // Construct a matrix to transform from tangent to world space
   mat3 tangentToWorldMat = mat3(v_tangent,
                                 v_binormal,
                                 v_normal);
      // Transform normal to world space and normalize
   normal = normalize(tangentToWorldMat * normal);

   // Normalize the light direction
   vec3 lightDirection = normalize(v_lightDirection);

   // Compute N.L
   float nDotL = dot(normal, lightDirection);

   // Compute reflection vector
   vec3 reflection = (2.0 * normal * nDotL) - lightDirection;
```

```
    // Use the reflection vector to fetch from the environment map
    vec4 envColor = textureCube(s_envMap, reflection);

    // Output final color
    gl_FragColor = 0.25 * baseColor + envColor;
}
```

片段着色器里，注意到从法线映射获取法线矢量的方法和获取每片段光照的方法相同。不同是替代转变法线矢量到切线空间，而是转变法线矢量到世界空间。使用 tangentToWorld 结果的 v_tangent,v_binormal, 和 v_normal 矢量乘法线矢量获取最后结果。反射矢量最后通过世界空间的光照方向矢量和法线矢量计算得到。在世界空间计算出的发射矢量恰好是我们需要的立方体贴图的环境映射。这个矢量使用 reflection 矢量做贴图坐标调用 textureCube 函数完成。最后计算的结果 gl_FragColor 作为背景映射颜色和环境映射颜色的一个组成。背景颜色在这个例子中乘 0.25，为了让环境映射更清楚可见。

这个例子显示了基本的环境映射。基本的技术实现了非常大的效果。附加的技术可以使用菲涅尔方法衰减反射得到更好的材料反射光模型。前面提到，另一种普遍采用的技术是动态渲染一个立方体场景，当场景改变时环境映射也发生变化。使用这里我们教你的基本技术，你能衍生其他更好的反射效果的技术。

使用点纹理的粒子系统

下一个例子是使用点纹理渲染粒子爆炸。例子的目的是显示顶点着色器的粒子动画和使用粒子纹理渲染粒子。我们讲的例子在 Chapter_13/ParticleSystem，结果看图 13-3

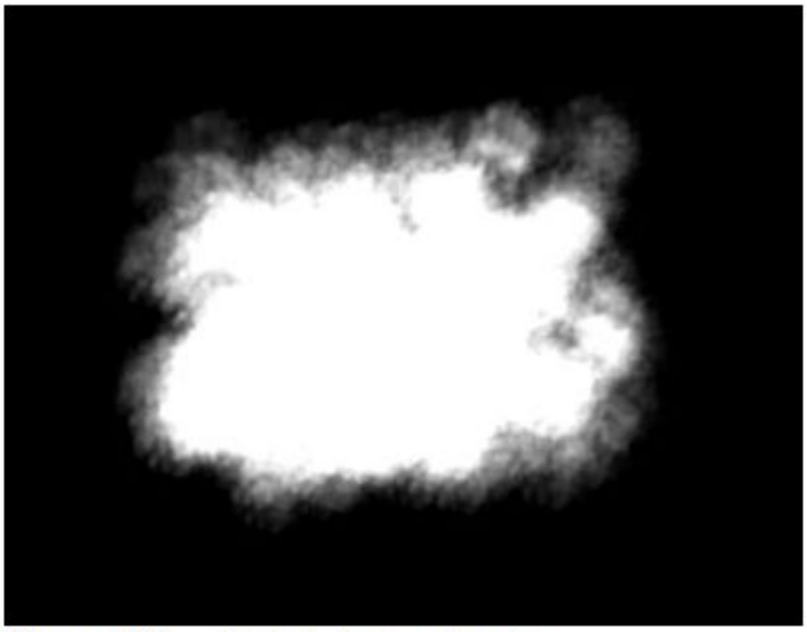


Figure 13-3 Particle System Sample

粒子系统建立

进入例子代码前，我们想说一下例子使用的方法。例子的目标是不使用 CPU 动态的修改顶点，渲染一个例子爆炸。也就是说除了 uniform 变量，当渲染爆炸时不改变任何顶点数据。为完成这个目标，着色器有一些输入。

初始化阶段，程序要初始化顶点矩阵里的这些值，每个粒子都需要，基于随机的值：

生命周期—粒子的生命时间，单位秒。

开始的位置—爆炸开始时粒子的位置

结束位置—粒子爆炸的结束位置（粒子在开始和结束位置之间动态线性插补。）

每个爆炸有几个 uniforms 全局设置：

中心位置—爆炸中心（每个顶点位置依此为这个中心做偏置）

颜色—爆炸的轮廓颜色

时间—当前时间，单位秒

顶点着色器的粒子系统

有了上面的信息，顶点和片段就完全能够渲染粒子系统的移动、跌落。让我们开始顶点着色器代码。

例 13-5 顶点着色器粒子系统

```
uniform float u_time;
uniform vec3 u_centerPosition;
attribute float a_lifetime;
attribute vec3 a_startPosition;
attribute vec3 a_endPosition;
varying float v_lifetime;
void main()
{
   if(u_time <= a_lifetime)
   {
      gl_Position.xyz = a_startPosition + (u_time * a_endPosition);
      gl_Position.xyz += u_centerPosition;
      gl_Position.w = 1.0;
   }
   else
      gl_Position = vec4(-1000, -1000, 0, 0);
      v_lifetime = 1.0 - (u_time / a_lifetime);
      v_lifetime = clamp(v_lifetime, 0.0, 1.0);
      gl_PointSize = (v_lifetime * v_lifetime) * 40.0;
}
```

首先输入顶点着色器的是 uniform 变量 u_time，这个变量设定当前消耗的时间，单位是秒，当时间超过了一次爆炸的长度，它的值复位到 0.0。下一个输入顶点着色器的 uniform 变量是 u_centerPosition，这个值设定一个新爆炸开始时的中心地点。设置 u_time 和 u_centerPosition 的代码在 Update()函数中，看例 13-6.

例 13-6 粒子系统采样更新函数

```
void Update (ESContext *esContext, float deltaTime)
{
   UserData *userData = esContext->userData;
   userData->time += deltaTime;
   if(userData->time >= 1.0f)
   {
      float centerPos[3];
      float color[4];
      userData->time = 0.0f;
      // Pick a new start location and color
```

```
        centerPos[0] = ((float)(rand() % 10000) / 10000.0f) - 0.5f;
        centerPos[1] = ((float)(rand() % 10000) / 10000.0f) - 0.5f;
        centerPos[2] = ((float)(rand() % 10000) / 10000.0f) - 0.5f;

        glUniform3fv(userData->centerPositionLoc, 1, &centerPos[0]);
        // Random color
        color[0] = ((float)(rand() % 10000) / 20000.0f) + 0.5f;
        color[1] = ((float)(rand() % 10000) / 20000.0f) + 0.5f;
        color[2] = ((float)(rand() % 10000) / 20000.0f) + 0.5f;
        color[3] = 0.5;
        glUniform4fv(userData->colorLoc, 1, &color[0]);
    }
    // Load uniform time variable
    glUniform1f(userData->timeLoc, userData->time);
}
```

你可以看到,Update()函数每隔一秒复位时间,然后为第二次爆炸设定新的中心位置和时间。函数保持 u_time 变量每帧更新。

返回的顶点着色器，输入到顶点着色器的属性是例子生命周期，例子开始位置和结束位置。初始化工作是随意的寻找程序的 Init 函数值。顶点着色器的首先检查是否粒子生命已经终止，如果是 gl_Position 值被设定为(-1000,-1000),这是快速的方法让点离开屏幕。因为点被剪切，后面所有的点移动过程被跳过。如果粒子仍然存在，它的位置被设定为在开始位置和结束位置的线性插补值。下面顶点着色器使用变量 v_lifetime 传递还有生命周期的粒子到片段着色器。生命周期时间在片段着色器中消耗到结束。顶点着色器设定点的位置基于粒子剩余的存活时间即内置的 gl_PointSize 变量。当粒子到了它们生命周期结束时，所有的影响消失。

片段着色器粒子系统

片段着色器的例子看例 13-7

例 13-7 片段着色器粒子系统

```
precision mediump float;
uniform vec4 u_color;
varying float v_lifetime;
uniform sampler2D s_texture;
void main()
{
    vec4 texColor;
    texColor = texture2D(s_texture, gl_PointCoord);
    gl_FragColor = vec4(u_color) * texColor;
    gl_FragColor.a *= v_lifetime;
}
```

首先输入片段着色器的 uniform 变量是 u_color。它在爆炸开始时使用 Update 设置。下一步，顶点着色器设置的变量 v_lifetime 在片段着色器里声明。它也是 2D 烟雾贴图图像绑定的采样器声明。

片段着色器相对是简单的。贴图使用 gl_PointCoord 变量做贴图坐标。这是粒子系统设定点纹理边缘的固定值（第 7 章基元装配和光栅化的绘制基元里描述）。片段着色器里如果

要求旋转点，片段着色器能扩展旋转点的坐标。这需要额外的片段着色器指令，但不减少点纹理的灵活性。

贴图颜色随 u_color 变量减弱，透明度随粒子生命周期减弱。应用程序使用下面的行为使能透明度混合。

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

这样做的结果是片段着色器里的透明度可以随着颜色调节。这个值被加入到片段着色器存储的目的地。对粒子系统增加混合的效果。注意对不同的粒子效果应该使用不同的透明度混合模式来完成。

实现绘制粒子系统的代码看例 13-8

例 13-8 粒子系统采样绘制行为

```
void Draw(ESContext *esContext)
{
   UserData *userData = esContext->userData;

   // Set the viewport
   glViewport(0, 0, esContext->width, esContext->height);

   // Clear the color buffer
   glClear(GL_COLOR_BUFFER_BIT);
   // Use the program object
   glUseProgram(userData->programObject);
   // Load the vertex attributes
   glVertexAttribPointer(userData->lifetimeLoc, 1, GL_FLOAT,
                         GL_FALSE, PARTICLE_SIZE * sizeof(GLfloat),
                         userData->particleData);
      glVertexAttribPointer(userData->endPositionLoc, 3, GL_FLOAT,
                         GL_FALSE, PARTICLE_SIZE * sizeof(GLfloat),
                         &userData->particleData[1]);
   glVertexAttribPointer(userData->startPositionLoc, 3, GL_FLOAT,
                         GL_FALSE, PARTICLE_SIZE * sizeof(GLfloat),
                         &userData->particleData[4]);

   glEnableVertexAttribArray(userData->lifetimeLoc);
   glEnableVertexAttribArray(userData->endPositionLoc);
   glEnableVertexAttribArray(userData->startPositionLoc);

   // Blend particles
   glEnable(GL_BLEND);
   glBlendFunc(GL_SRC_ALPHA, GL_ONE);
   // Bind the texture
   glActiveTexture(GL_TEXTURE0);
   glBindTexture(GL_TEXTURE_2D, userData->textureId);
   glEnable(GL_TEXTURE_2D);
```

```
    // Set the sampler texture unit to 0
    glUniform1i(userData->samplerLoc, 0);
    glDrawArrays(GL_POINTS, 0, NUM_PARTICLES);

    eglSwapBuffers(esContext->eglDisplay, esContext->eglSurface);
}
```

绘制函数，从设定视口和清除屏幕开始。然后使用 glVertexAttribPointer 选择项目对象和装载顶点数据。因为顶点矩阵值没有改变，例子是顶点缓冲区对象而不是客户区顶点矩阵。一般情况下要求顶点数据不改变，这样会减少顶点带宽的使用。在例子中顶点缓冲区对象没使用，让代码更简洁。设置完顶点矩阵，使能混合行为，绑定烟雾贴图，使用 glDrawArrays 绘制粒子。

不像三角形，对粒子纹理来说没有联系，使用 glDrawElements 渲染粒子不会带来什么好处。但粒子系统经常需要存储深度从后到前去得到合适的透明度混合结果。在这种情况下，修改元素矩阵的绘图顺序的方法被使用。这是有效率的方法，因为这样每帧需要最小的带宽（仅仅索引数据需要改变，对顶点数据几乎没有改变）。

例子显示了一些粒子系统使用粒子纹理的技术。在整个 GPU 的顶点着色器里粒子系统是有活性的。粒子的尺寸使用 gl_PointSize 变量基于粒子生命周期减弱。另外，粒子使用 gl_PointCoord 建立的贴图坐标变量被渲染。OpenGL ES 2.0 在粒子系统里那是需要执行的基本的元素。

图像后处理

下一个粒子我们阐述图像后处理。联合使用帧缓冲区对象和着色器，可能执行多种图像后处理技术。第一个例子是 Bloom 效果，代码看 Chapter_13/PostProcess，结果看图像 13-4。

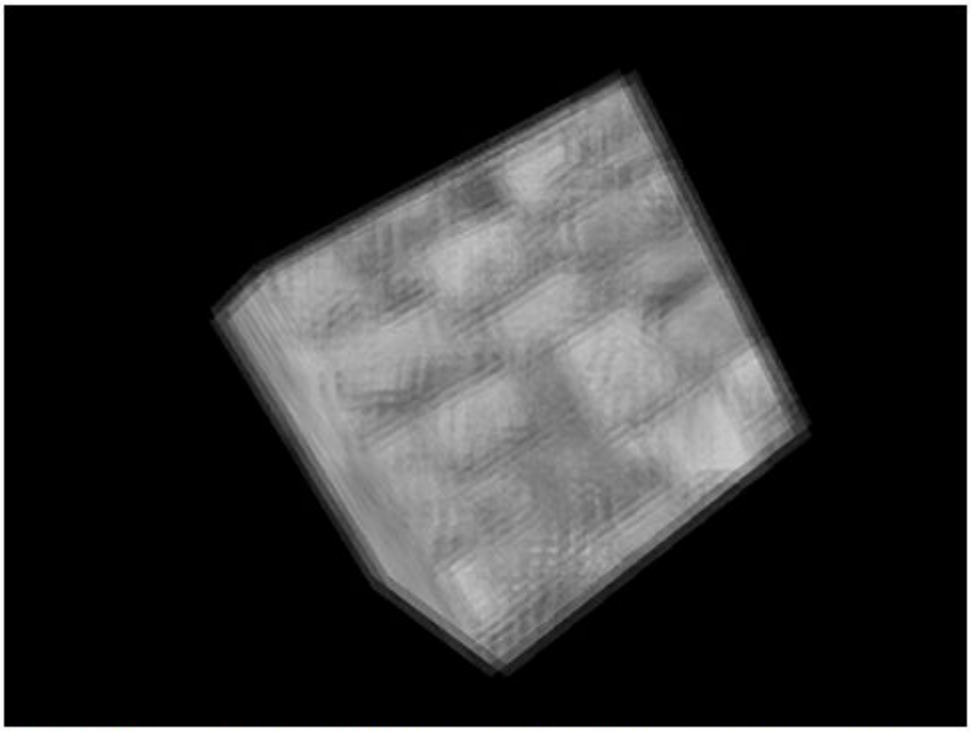


Figure 13-4 Image Postprocessing Example (see Color Plate 6)

纹理烘焙设置

这个例子渲染一个立方体贴图到帧缓冲区对象，然后使用颜色附着做贴图后处理。使用

渲染的贴图做源，全屏幕四方体被渲染到屏幕。一个片段着色器运行覆盖整个屏幕，执行模糊过滤。一般情况下，后处理技术能被使用下面的模式完成。

渲染场景到非显示帧缓冲区（FBO）

绑定 FBO 贴图做源渲染整个四方屏幕

执行片段着色器对四方形执行过滤

一些算法要求对一个图像多次执行，一些要求很多难懂的输入。然而一般思想是使用片段着色器对整个屏幕四方形执行后处理算法。

模糊片段着色器

片段着色器使用整屏模糊例子看例 13-9.

例 13-9 片段着色器模糊

```
precision mediump float;
uniform sampler2D renderTexture;
varying vec2 v_texCoord;
uniform float u_blurStep;
void main(void)
{
   vec4 sample0,
        sample1,
        sample2,
        sample3;

   float step = u_blurStep / 100.0;

   sample0 = texture2D(renderTexture,
                       vec2(v_texCoord.x - step,
                            v_texCoord.y - step));
   sample1 = texture2D(renderTexture,
                       vec2(v_texCoord.x + step,
                            v_texCoord.y + step));
   sample2 = texture2D(renderTexture,
                       vec2(v_texCoord.x + step,
                            v_texCoord.y - step));
   sample3 = texture2D(renderTexture,
                       vec2(v_texCoord.x - step,
                            v_texCoord.y + step));

   gl_FragColor = (sample0 + sample1 + sample2 + sample3) / 4.0;
}
```

着色器开始基于 uniform 变量 u_blurStep 计算 step 变量。Step 被用于在获取图像采样时，计算贴图坐标的偏置。四个不同的采样从图像中获取，在着色器里平均。Step 是四个方向上的贴图坐标偏置，四个采样是从中心向对角线方向。Step 值越大，图像越模糊。一个可能的优化是在顶点着色器里计算贴图坐标配置以变量形式输入到片段着色器。这将减少片段着色器的计算。

光线绽放

现在我们看了简单的图像后处理过程，让我们看一个更复杂的例子。使用前面介绍的模糊技术，我们可以执行光线绽放的效果。光线绽放在眼睛在黑暗处突然看到一个亮光时发生。黑暗中的光线刺痛了眼睛的效果。例子看 Chapter_13/PostProcess，图片结果看 13-5。

从屏幕截图中我们能看到，黑色背景下一个汽车的模型，算法如下：

1. 清除非显示缓冲区渲染目标(rt0)，在黑色背景下绘制物体
2. 使用模糊步骤 1.0 模糊非显示区目标(rt0)和另一个渲染目标(rt1)。
3. 使用模糊步骤 2.0 模糊非显示区目标(rt1)和另一个渲染目标(rt0)。
4. 注意：为更多模糊，重复步骤 2 和 3，增加模糊步骤。
5. 在后缓冲区渲物体
6. 和后缓冲区混合最后的渲染目标

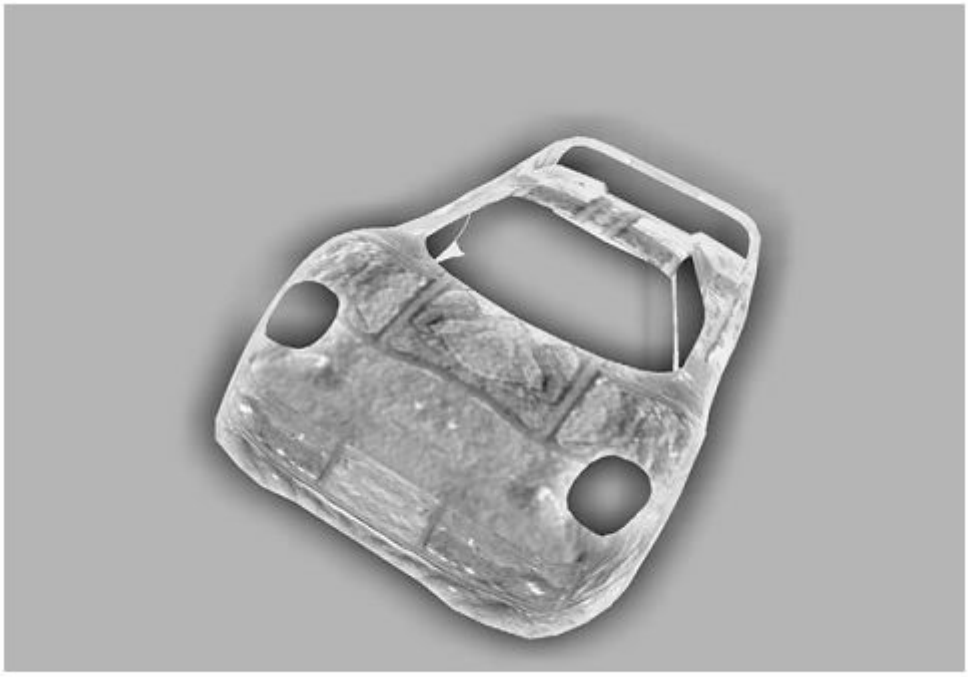


Figure 13-5 Light Bloom Effect

算法的图解看图 13-6，显示了产生最后图像的每一步。

就像你在图 13-6 看到的，首先用黑色渲染目标。在下一步目标和第二个渲染目标模糊。模糊过后的目标然后再和扩展的模糊躯干放回原始目标。最后，渲染目标和原始目标混合。使用两个目标成功模糊的观念常常暗指 pingponging。光线绽放的数目能增加，只要使用模糊的 pingponging 次数增加。模糊操作的着色器代码和前面例子是一样的。仅仅的区别是每次增加的模糊步骤。

有大量的算法能执行 FBOs 和着色器的联合。其它普遍使用的技术包括色调映射、选择性模糊、扭曲、场景切换、深度视野等等，使用我们展示给你的技术，你能开始使用着色器增强后处理算法。

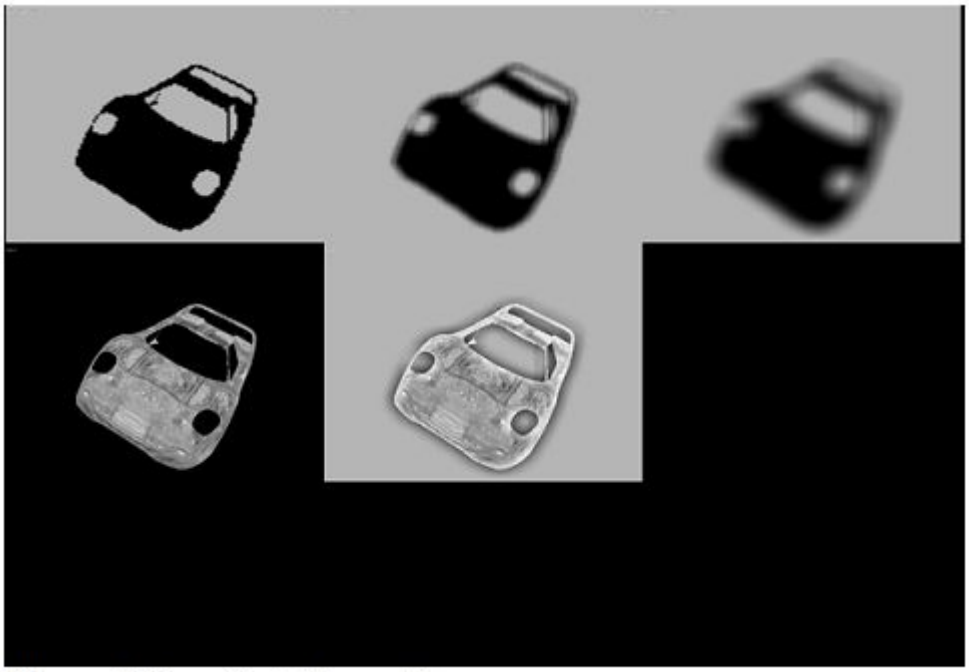


Figure 13-6 Light Bloom Stages

阴影贴图

一个普遍使用的技术像阴影映射和使用阴影贴图的反光。介绍使用阴影贴图前，我们先介绍渲染聚光灯阴影。大多数使用阴影贴图数学计算上的复杂是计算应用贴图坐标。我们显示给你的方法同样适用于阴影映射和反射。聚光灯阴影的例子看 Chapter_13/ProjectiveSpotlight，图像看图 13-7.

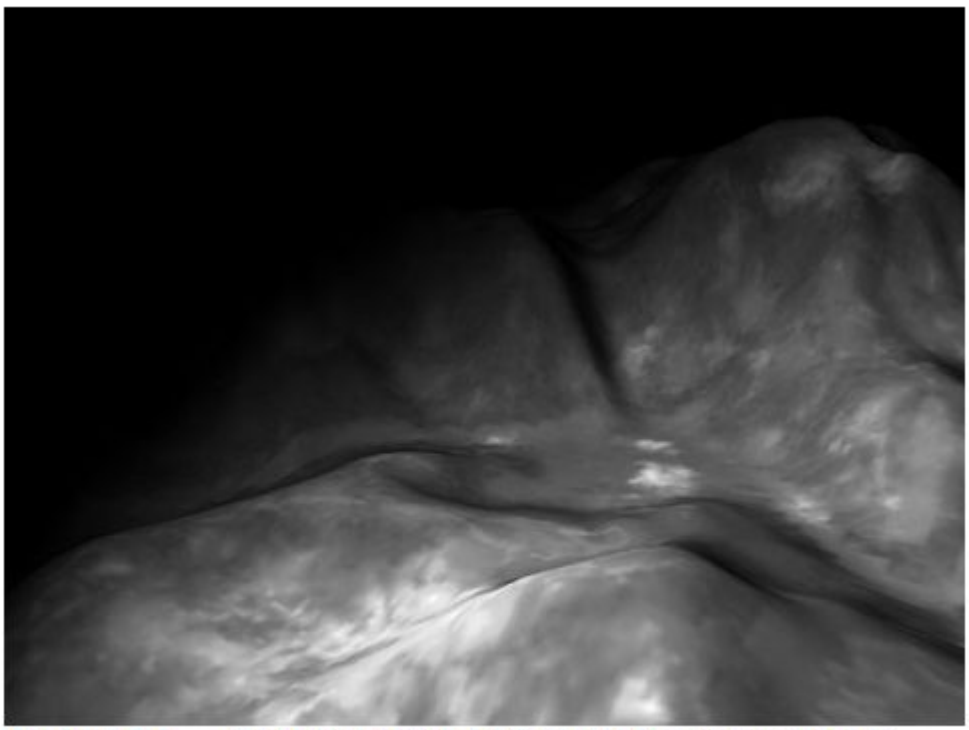


Figure 13-7 Projective Spotlight Example (see Color Plate 7)

阴影贴图基础

图 13-8 例子使用 2D 贴图图像，在几何地形的表面使用了阴影贴图。聚光灯阴影是 GPU 引进着色器前，估算聚光灯像素衰减普遍使用的技术。因为高效率，聚光灯阴影仍然是有效的解决方案。使用聚光灯阴影只需要在片段着色器里执行一个单独的贴图在顶点着色器做一些设置。另外 2D 贴图图像能投射很多真实的图像，能够完成很多可能的效果。

通过贴图投射我们能做什么呢？最基本的是投射 3D 贴图坐标成为 2D 贴图图像。(s, t)坐标和(r)坐标分离，使用(s/r, t/r)得到一个纹理。OpenGL ES 着色器语言里做投射贴图的函数是 texture2Dproj。

```
vec4    texture2DProj(sampler2D sampler, vec3 coord[,
                      float bias])
sampler     a sampler bound to a texture unit specifying the texture to
            fetch from
coord       a 3D texture coordinate used to fetch from the texture map.
            The (x, y) arguments are divided by (z) such that the fetch
            occurs at (x/z, y/z)
bias        an optional LOD bias to apply
```

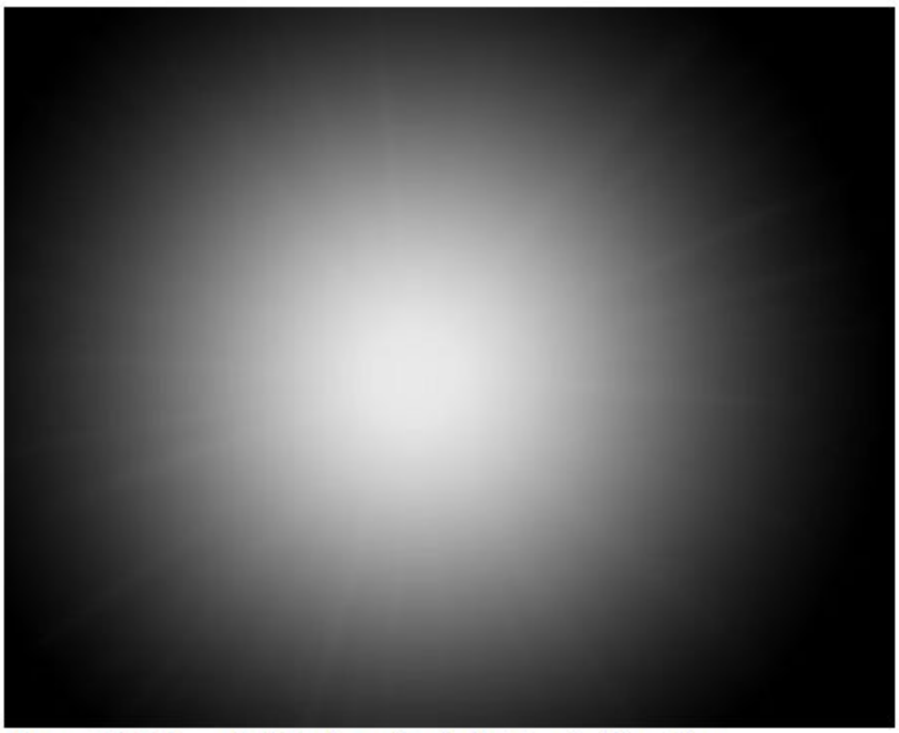


Figure 13-8 2D Texture Projected onto Terrain

投影光的主意是传递物体的位置到光线的投影空间。投影到光空间的位置，被测量和偏置后，作为投影贴图坐标使用。RenderMonkey 的例子中的顶点着色器做传递位置值到光投影空间的工作。

投影贴图矩阵

有三个矩阵用于传递位置到光线投影空间，得到投影贴图坐标。

光投影—有关光源的投影矩阵，使用视图、宽高比和光的远近平面

光视图—光源的视图矩阵，把光源当作照相机建立的

偏移矩阵—转换光源空间投射位置到 3D 投射贴图位置的矩阵

光源投影矩阵将像其它投影矩阵一样使用光源可视区域（FOV）的光源参数、宽高

比(aspect)、近(zNear)远(zFar)平面距离构造。

$$\begin{bmatrix} \frac{\cot\left(\frac{FOV}{2}\right)}{aspect} & 0 & 0 & 0 \\ 0 & \cot\left(\frac{FOV}{2}\right) & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & \frac{2\times zFar\times zNear}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

光视图矩阵使用三个基本的轴距即光线视图轴和光线位置距离。我们的轴指右面、上面和观看的矢量。

$$\begin{bmatrix} right.x & up.x & look.x & 0 \\ right.y & up.y & look.y & 0 \\ right.z & up.z & look.z & 0 \\ dot(right, -lightPos) & dot(up, -lightPos) & dot(look, -lightPos) & 1 \end{bmatrix}$$

使用视图和投影矩阵转换物体位置后，我们必须转化坐标到投影贴图坐标。这使用光线投影空间的位置(x, y, z)的 3×3 偏移矩阵完成。偏移矩阵做线性转换，从范围[-1, 1]转换到范围[0, 1]。在范围[0, 1]内的值可以使用做贴图坐标。

$$\begin{bmatrix} 0.5 & 0.0 & 0.0 \\ 0.0 & -0.5 & 0.0 \\ 0.5 & 0.5 & 1.0 \end{bmatrix}$$

典型的,转化位置到投影坐标系将使用 CPU 联合投影、视图和偏移矩阵一起完成(使用 4×4 的偏移矩阵)。这将在顶点着色器里装载单一的 uniform 矩阵传递位置。例子里我们在顶点着色器里执行这个计算解释。

聚光灯阴影投射

现在我们看一下基本的机制，解释顶点着色器，看例 13-10.

例 13-10 顶点着色器投射贴图

```
uniform float u_time_0_X;
uniform mat4 u_matProjection;
uniform mat4 u_matViewProjection;
attribute vec4 a_vertex;
attribute vec2 a_texCoord0;
attribute vec3 a_normal;
varying vec2 v_texCoord;
varying vec3 v_projTexCoord;
```

```
varying vec3 v_normal;
varying vec3 v_lightDir;
void main(void)
{
   gl_Position = u_matViewProjection * a_vertex;
   v_texCoord = a_texCoord0.xy;
      // Compute a light position based on time
  vec3 lightPos;
  lightPos.x = cos(u_time_0_X);
  lightPos.z = sin(u_time_0_X);
  lightPos.xz = 100.0 * normalize(lightPos.xz);
  lightPos.y = 100.0;
  // Compute the light coordinate axes
  vec3 look = -normalize(lightPos);
  vec3 right = cross(vec3(0.0, 0.0, 1.0), look);
  vec3 up = cross(look, right);

  // Create a view matrix for the light
  mat4 lightView = mat4(right, dot(right, -lightPos),
                        up,    dot(up, -lightPos),
                        look,  dot(look, -lightPos),
                        0.0, 0.0, 0.0, 1.0);

  // Transform position into light view space
  vec4 objPosLight = a_vertex * lightView;
  // Transform position into projective light view space
  objPosLight = u_matProjection * objPosLight;
   // Create bias matrix
   mat3 biasMatrix = mat3(0.5,  0.0, 0.5,
                          0.0, -0.5, 0.5,
                          0.0,  0.0, 1.0);
   // Compute projective texture coordinates
   v_projTexCoord = objPosLight.xyz * biasMatrix;
   v_lightDir = normalize(a_vertex.xyz - lightPos);
   v_normal = a_normal;
}
```

着色器第一步使用 u_matViewProjection 做位置转换，基于 v_texCoord 变量为底图输出贴图坐标。着色器做的下一步是基于时间计算光的位置。这些代码能被忽略，但帮助顶点着色器产生光线的动画。典型应用中，这被 CPU 完成而不是着色器。

基于光源位置，顶点着色器计算三个坐标轴矢量：观看方向、右面和上面。这些矢量创建光线的视图矩阵，lightView 变量使用前面方程描述的值。物体的输入位置被 lightView 矩阵转换，转变到光线空间。下一步是使用透视矩阵转换光线空间位置到投射光线空间。不是创建新的透视矩阵，粒子使用照相机 u_matProjection 矩阵。典型的实际的应用常常创建基于聚光灯角度和距离创建自己的透视矩阵。

投影光线空间的位置 biasMatrix 创建后传递到投影贴图坐标。最后投影贴图坐标存储在 vec3 变量 v_projTexCoord 中。另外顶点着色器也传送光线方向和法线矢量到片段着色器的 v_lightDir 和 v_normal 变量。这决定了片段是面向光源或是远离光源。

片段着色器执行实际的投影贴图，应用聚光灯的投影贴图到物体表面。

例 13-11 片段着色器投影贴图

```
precision mediump float;
uniform sampler2D baseMap;
uniform sampler2D spotLight;
varying vec2 v_texCoord;
varying vec3 v_projTexCoord;
varying vec3 v_normal;
varying vec3 v_lightDir;
void main(void)
{
   // Projective fetch of spotlight
   vec4 spotLightColor = texture2DProj(spotLight, v_projTexCoord);
   // Basemap
   vec4 baseColor = texture2D(baseMap, v_texCoord);

   // Compute N.L
   float nDotL = max(0.0, -dot(v_normal, v_lightDir));

   gl_FragColor = spotLightColor * baseColor * 2.0 * nDotL;
}
```

片段着色器执行的第一步是使用 texture2Dproj 获取投影贴图。投影贴图坐标在顶点着色器里计算，传递到用来计算投影贴图的 v_projTexCoord 变量。投影贴图的包装模式被设置为 GL_CLAMP_TO_EDGE，min/mag 过滤设置为 GL_LINEAR。片段着色器使用 v_texCoord 变量获取底图的颜色。下一步着色器计算光线方向和法线之间的点。在片段远离光源聚光灯投影不能应用时，减弱最终的颜色。最后所有的成分被乘到一起（被 2.0 标准化，减少亮度）。这给我们最终的图 13-7 投影聚光灯地形图像。

本章的开始谈到，这个例子重要的是计算投影贴图坐标的函数。计算和产生阴影映射是非常类似的。同样，渲染投影贴图的反射，需要你传递相机反射位置到投影视图空间。你可以同样方法来做，只不过替换反射相机矩阵为光线矩阵。投影矩阵是实现先进效果的有力工具，你应该基本懂得怎么使用它。

使用 3D 贴图噪声

下面的渲染技术我们称为 3D 贴图噪声。第 9 章，介绍了基本的 3D 贴图。回忆一下，3D 贴图本质是 2D 贴图切片的堆积，有很多可能使用 3D 贴图，其中一个可能是表达噪声。这部分我们显示一个例子使用 3D 大量的噪声产生一个薄雾效果。这个例子基于第 10 章片段着色器线性雾。例子看 RenderMonkey 工作区的 Chapter_13/Noise3D，图片看图 13-9.

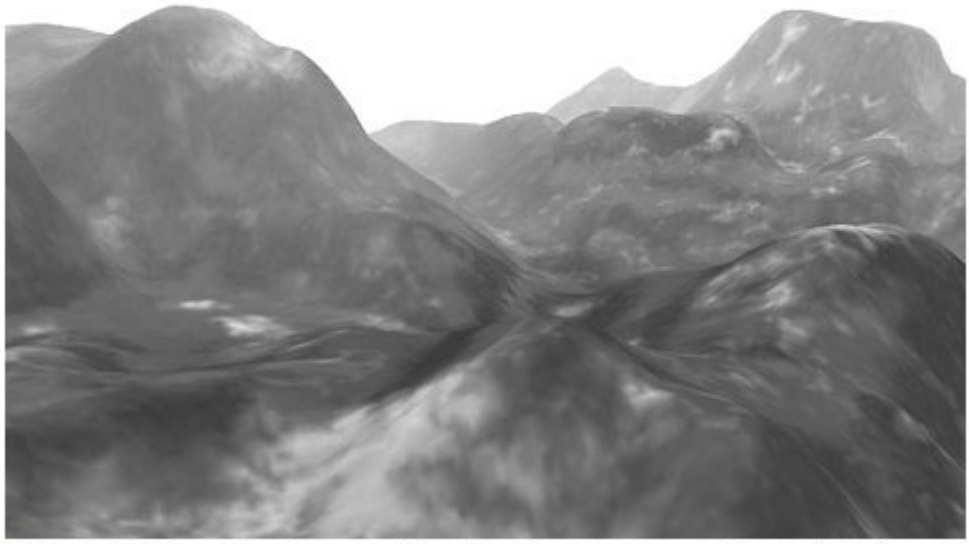


Figure 13-9 Fog Distorted by 3D Noise Texture (see Color Plate 8)

创造噪声

应用噪声是非常普遍的技术在大量的 3D 效果中。OpenGL 着色器编程语言（不是 OpenGL ES 着色器编程语言）能在一维、二维、三位或者四维上计算噪声。这种方法返回大量随机的连续的噪声值基于可重复的输入值。问题是这些计算是非常费时的。大多数的 GPU 不使用硬件执行噪声行为，而是使用着色器执行（或者使用软件在 CPU 上执行，效果更糟）。这有很多着色器指令能够增强噪声行为。使用片段着色器大多数时间将太慢，这是 OpenGL ES 工作组决定删除着色器语言的噪声函数（厂商仍然可以通过扩展支持）。

虽然在片段着色器里计算噪声可能很珍贵，但我们可以使用 3D 贴图解决这个问题。通过预计算噪声，把结果放到 3D 贴图里，能够产生一个能接受质量的噪声。有很多算法能产生噪声，本章结尾有一个产生各种噪声算法的参考列表和链接。我们讨论产生晶格梯度噪声的算法。这是普遍使用的产生噪声的方法。例如一个晶格梯度的噪声被强大的着色器语言噪声函数产生。

梯度噪声算法使用 3D 坐标做输入返回一个浮点噪声值。为使用输入的(x,y,z)产生噪声值，我们映射 x,y,z 值到适当晶格的整数位置。晶格的单位是可编程，我们程序是 256 单元。在晶格的每个单元，我们产生和存储随机的梯度矢量。例 13-12 描述了我们产生的梯度矢量。

例 13-12 产生梯度矢量

```
// permTable describes a random permutation of
// 8-bit values from 0 to 255.
static unsigned char   permTable[256] = {
   0xE1, 0x9B, 0xD2, 0x6C, 0xAF, 0xC7, 0xDD, 0x90,
   0xCB, 0x74, 0x46, 0xD5, 0x45, 0x9E, 0x21, 0xFC,
   0x05, 0x52, 0xAD, 0x85, 0xDE, 0x8B, 0xAE, 0x1B,
```

```
    0x09, 0x47, 0x5A, 0xF6, 0x4B, 0x82, 0x5B, 0xBF,
    0xA9, 0x8A, 0x02, 0x97, 0xC2, 0xEB, 0x51, 0x07,
    0x19, 0x71, 0xE4, 0x9F, 0xCD, 0xFD, 0x86, 0x8E,
    0xF8, 0x41, 0xE0, 0xD9, 0x16, 0x79, 0xE5, 0x3F,
    0x59, 0x67, 0x60, 0x68, 0x9C, 0x11, 0xC9, 0x81,
    0x24, 0x08, 0xA5, 0x6E, 0xED, 0x75, 0xE7, 0x38,
    0x84, 0xD3, 0x98, 0x14, 0xB5, 0x6F, 0xEF, 0xDA,
    0xAA, 0xA3, 0x33, 0xAC, 0x9D, 0x2F, 0x50, 0xD4,
    0xB0, 0xFA, 0x57, 0x31, 0x63, 0xF2, 0x88, 0xBD,
    0xA2, 0x73, 0x2C, 0x2B, 0x7C, 0x5E, 0x96, 0x10,
    0x8D, 0xF7, 0x20, 0x0A, 0xC6, 0xDF, 0xFF, 0x48,
    0x35, 0x83, 0x54, 0x39, 0xDC, 0xC5, 0x3A, 0x32,
    0xD0, 0x0B, 0xF1, 0x1C, 0x03, 0xC0, 0x3E, 0xCA,
    0x12, 0xD7, 0x99, 0x18, 0x4C, 0x29, 0x0F, 0xB3,
    0x27, 0x2E, 0x37, 0x06, 0x80, 0xA7, 0x17, 0xBC,
    0x6A, 0x22, 0xBB, 0x8C, 0xA4, 0x49, 0x70, 0xB6,
    0xF4, 0xC3, 0xE3, 0x0D, 0x23, 0x4D, 0xC4, 0xB9,
    0x1A, 0xC8, 0xE2, 0x77, 0x1F, 0x7B, 0xA8, 0x7D,
    0xF9, 0x44, 0xB7, 0xE6, 0xB1, 0x87, 0xA0, 0xB4,
    0x0C, 0x01, 0xF3, 0x94, 0x66, 0xA6, 0x26, 0xEE,
    0xFB, 0x25, 0xF0, 0x7E, 0x40, 0x4A, 0xA1, 0x28,
    0xB8, 0x95, 0xAB, 0xB2, 0x65, 0x42, 0x1D, 0x3B,
    0x92, 0x3D, 0xFE, 0x6B, 0x2A, 0x56, 0x9A, 0x04,
    0xEC, 0xE8, 0x78, 0x15, 0xE9, 0xD1, 0x2D, 0x62,
    0xC1, 0x72, 0x4E, 0x13, 0xCE, 0x0E, 0x76, 0x7F,
    0x30, 0x4F, 0x93, 0x55, 0x1E, 0xCF, 0xDB, 0x36,
    0x58, 0xEA, 0xBE, 0x7A, 0x5F, 0x43, 0x8F, 0x6D,
    0x89, 0xD6, 0x91, 0x5D, 0x5C, 0x64, 0xF5, 0x00,
    0xD8, 0xBA, 0x3C, 0x53, 0x69, 0x61, 0xCC, 0x34,
};
#define NOISE_TABLE_MASK   255
// lattice gradients 3D noise
static float gradientTable[256*3];
#define FLOOR(x)       ((int)(x) - ((x) < 0 && (x) != (int)(x)))
#define smoothstep(t)  (t * t * (3.0f - 2.0f * t))
#define lerp(t, a, b)  (a + t * (b - a))
void
initNoiseTable()
{
   long          rnd;
   int           i;
   double        a;
   float         x, y, z, r, theta;
   float         gradients[256*3];
```

```
    unsigned int  *p, *psrc;
    srandom(0);
    // build gradient table for 3D noise
    for(i=0; i<256; i++)
    {
       // calculate 1 - 2 * random number
       rnd = random();
       a = (random() & 0x7FFFFFFF) / (double) 0x7FFFFFFF;
       z = (float)(1.0 - 2.0 * a);
       r = (float)sqrt(1.0 - z * z);   // r is radius of circle
       rnd = random();
       a = (float)((random() & 0x7FFFFFFF) / (double) 0x7FFFFFFF);
       theta = (float)(2.0 * M_PI * a);
       x = (float)(r * (float)cos(a));
       y = (float)(r * (float)sin(a));
       gradients[i*3] = x;
       gradients[i*3+1] = y;
       gradients[i*3+2] = z;
    }
    // use the index in the permutation table to load the
    // gradient values from gradients to gradientTable
    p = (unsigned int *)gradientTable;
    psrc = (unsigned int *)gradients;
    for (i=0; i<256; i++)
    {
       int indx = permTable[i];
       p[i*3] = psrc[indx*3];
       p[i*3+1] = psrc[indx*3+1];
       p[i*3+2] = psrc[indx*3+2];
    }
}
```

例 13-13 描述使用随机的梯度矢量和输入 3D 坐标产生梯度噪声。

例 13-13 3D 噪声

```
//
// generate the value of gradient noise for a given lattice point
//
// (ix, iy, iz) specifies the 3D lattice position
// (fx, fy, fz) specifies the fractional part
//
static float
glattice3D(int ix, int iy, int iz, float fx, float fy, float fz)
{
    float   *g;
    int      indx, y, z;
```

```
    z = permTable[iz & NOISE_TABLE_MASK];
    y = permTable[(iy + z) & NOISE_TABLE_MASK];
    indx = (ix + y) & NOISE_TABLE_MASK;
    g = &gradientTable[indx*3];
    return (g[0]*fx + g[1]*fy + g[2]*fz);
}
//
// generate the 3D noise value
// f describes input (x, y, z) position for which the noise value
//  needs to be computed. noise3D returns the scalar noise value
//
float
noise3D(float *f)
{
    int   ix, iy, iz;
    float   fx0, fx1, fy0, fy1, fz0, fz1;
    float   wx, wy, wz;
    float   vx0, vx1, vy0, vy1, vz0, vz1;
    ix = FLOOR(f[0]);
    fx0 = f[0] - ix;
    fx1 = fx0 - 1;
    wx = smoothstep(fx0);
    iy = FLOOR(f[1]);
    fy0 = f[1] - iy;
    fy1 = fy0 - 1;
    wy = smoothstep(fy0);
    iz = FLOOR(f[2]);
    fz0 = f[2] - iz;
    fz1 = fz0 - 1;
    wz = smoothstep(fz0);
    vx0 = glattice3D(ix, iy, iz, fx0, fy0, fz0);
    vx1 = glattice3D(ix+1, iy, iz, fx1, fy0, fz0);
    vy0 = lerp(wx, vx0, vx1);
    vx0 = glattice3D(ix, iy+1, iz, fx0, fy1, fz0);
    vx1 = glattice3D(ix+1, iy+1, iz, fx1, fy1, fz0);
    vy1 = lerp(wx, vx0, vx1);
    vz0 = lerp(wy, vy0, vy1);

    vx0 = glattice3D(ix, iy, iz+1, fx0, fy0, fz1);
    vx1 = glattice3D(ix+1, iy, iz+1, fx1, fy0, fz1);
    vy0 = lerp(wx, vx0, vx1);
    vx0 = glattice3D(ix, iy+1, iz+1, fx0, fy1, fz1);
    vx1 = glattice3D(ix+1, iy+1, iz+1, fx1, fy1, fz1);
    vy1 = lerp(wx, vx0, vx1);
```

```
    vz1 = lerp(wy, vy0, vy1);
    return lerp(wz, vz0, vz1);;
}
```

3D 噪声返回值范围是[-1.0,1]。在整数梯度点上噪声值是 0。之间的点，三线性插补 8 个整数晶格点梯度值产生的点被用来产生标量噪声。图 13-10 显示了使用先前算法产生的梯度噪声的 2D 切片。

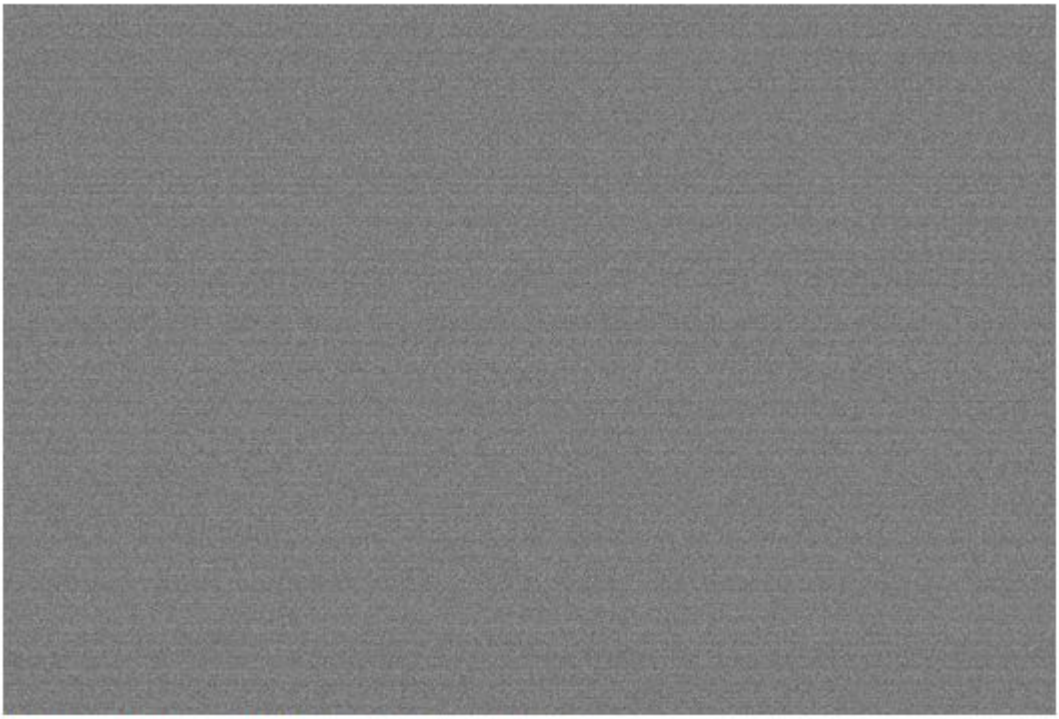


Figure 13-10 2D Slice of Gradient Noise

使用噪声

一旦我们产生了 3D 噪声卷，很容易使用它产生各种不同的影响。例子中是细微的雾。原理很简单：在三个方向上基于时间铺设 3D 噪声贴图，从贴图到扭曲雾因子。让我们看例 13-14 片段着色器的例子。

例 13-14 片段着色器噪声扭曲雾

```
#extension GL_OES_texture_3D : enable
precision mediump float;
uniform vec4 u_fogColor;
uniform float u_fogMaxDist;
uniform float u_fogMinDist;
uniform float u_time;
uniform sampler2D baseMap;
uniform sampler3D noiseVolume;
varying vec2 v_texCoord;
varying float v_eyeDist;
float computeLinearFogFactor()
{
    float factor;
```

```
    // Compute linear fog equation
    factor = (u_fogMaxDist - v_eyeDist) /
             (u_fogMaxDist - u_fogMinDist);

    return factor;
}
void main(void)
{
    float fogFactor = computeLinearFogFactor();
    vec4 baseColor = texture2D(baseMap, v_texCoord);

    // Distort fog factor by noise
    vec3 noiseCoord;
    noiseCoord.xy = v_texCoord.xy - (u_time * 0.1);
    noiseCoord.z = u_time * 0.1;

    fogFactor += texture3D(noiseVolume, noiseCoord).r;
    fogFactor = clamp(fogFactor, 0.0, 1.0);

    // Compute final color as a lerp with fog factor
    gl_FragColor = baseColor * fogFactor +
                   u_fogColor * (1.0 - fogFactor);
}
```

着色器做的第一件事是使用#extension 机制使能 3D 贴图扩展。如果应用不支持 GL_OES_texture_3D 扩展，着色器编译失败。这个着色器非常类似于第 10 章线性雾的例子。基本的不同是线性雾因子被 3D 噪声贴图扭曲。着色器计算 3D 贴图坐标基于时间和在 noiseCoord 中的位置。u_time 变量代表当前时间，每帧更新。3D 贴图的 s, t, 和 r 使用 GL_REPEAT 包装模式，噪声在表面是平滑的。(s, t)坐标基于基本的贴图坐标在两个方向弯曲，(r)坐标基于时间，他连续的弯曲。

3D 贴图是单通道(GL_LUMINANCE)的，所有贴图的红色被使用（绿色和蓝色和红色值相同）从卷获取的值也用于计算 fogFactor,，然后在基本颜色和雾颜色之间线性插补。结果是薄薄的雾出现在小山顶卷动。当变换 3D 贴图的坐标时，通过对 u_time 变量的测量，雾的速度能被容易的增加。

通过使用 3D 贴图产生噪声，能产生很多不同的影响。一些使用噪声的例子包括光柱里的灰尘、程序贴图产生自然界的效果、模拟水浪花。使用 3D 贴图是很好的方法节省硬件，产生高质量的视觉效果。不像使用昂贵的硬件在着色器产生噪声，这个方法好高效率以致程序能运行在高的帧率。预先计算的噪声量是非常有价值的技巧让你的工具产生效果。

可编程纹理

下一个题目是可编程纹理。纹理典型描述的是 2D 图像、立方体、3D 图像。这些图像存储颜色和深度值。OpenGL ES 着色器颜色内置的函数做纹理坐标、称为采样器的纹理对象、返回颜色和深度值。可编程纹理指用程序产生的纹理而不是一个图片。指使用输入产生纹理颜色和深度值。

可编程纹理有下面的益处。

对比图片纹理更简洁的存储。你要存储的是程序代码，对比图片尺寸更小。

不像存储图片，程序纹理没有固定的解决，它们能不丢失细节的应用于物体表面。这意味着移动缩放物体时不用减少细节。而使用图像纹理将会遇到这些问题，因为它是固定的解决。

可编程纹理不利的方面有：

可编程纹理要求的函数可能在片段着色器上因为片段着色器代码的限制不能编译。OpenGL ES 2.0 运行在手持和嵌入式设备上，能用来产生不错纹理的算法的限制可能是个严重的问题。这个问题在手持设备上更严重，因为手持设备的绘图能力限制。

虽然可编程纹理有小的存储体积，但对比查找存储好的图片，可编程纹理可能使用多个循环来计算最终的纹理。你需要的带宽可能更高。考虑到指令执行和带宽的需求，你必须仔细选择。

循环可能很难实现，算法精度的不同，不同的 OpenGL ES 2.0 实现的内置函数不同，能产生不同的待处理问题。

可编程纹理有严重的锯齿痕迹。大多数的痕迹能够被处理但需要更多的代码，这加重了着色器的执行难度。

考虑是否使用可编程纹理还是存储一个图片，需要认真考虑程序的执行和要求的带宽。

可编程纹理例子

我们现在看一个展示可编程纹理的例子。我们非常熟悉在物体上绘制西洋棋图像。现在我们来看这个渲染。例子看 Chapter_13/ProceduralTextures 的 Checker.rfx。例 13-15 和 13-16 顶点和片段着色器执行西洋棋贴图程序。

例 13-15 检查顶点着色器

```
uniform mat4    mvp_matrix;   // combined modelview + projection
matrix
attribute vec4  a_position;   // input vertex position
attribute vec2  a_st;         // input texture coordinate
varying vec2    v_st;         // output texture coordinate
void
main()
{
   v_st = a_st;
   gl_Position = mvp_matrix * a_position;
}
```

在例 13-15 中的代码是很简单的。它联合使用模型视图和投影矩阵变换位置，使用变量(v_st) 传递纹理坐标(a_st)到片段着色器。

例 13-16 检查片段着色器

```
#ifdef GL_ES
precision highp float;
#endif
// frequency of the checkerboard pattern
uniform int     u_frequency;
// alternate colors that make the checkerboard pattern
uniform vec4    u_color0;
```

```
uniform vec4    u_color1;
varying vec2    v_st;
void
main()
{
   vec2    tcmod = mod(v_st * float(u_frequency), 1.0);
       if(tcmod.s < 0.5)
   {
      if(tcmod.t < 0.5)
         gl_FragColor = u_color1;
      else
         gl_FragColor = u_color0;
   }
   else
   {
      if(tcmod.t < 0.5)
         gl_FragColor = u_color0;
      else
         gl_FragColor = u_color1;
   }

   gl_FragColor = mix(color1, color0, delta);
}
```

例 13-16 的片段着色器代码使用 v_st 贴图坐标绘制贴图图案。很容易明白，片段着色器有弱的执行能力，因为多个条件的检查对并行执行的片段着色器是困难的。这也暗示使用 GPU 的顶点着色器和片段着色器并行执行的数目应该减少。例 13-17 片段着色器不再使用条件检查。

例 13-17 片段着色器检查

```
#ifdef GL_ES
precision highp float;
#endif
// frequency of the checkerboard pattern
uniform int     u_frequency;
// alternate colors that make the checkerboard pattern
uniform vec4    u_color0;
uniform vec4    u_color1;
varying vec2    v_st;
void
main()
{
   vec2   texcoord = mod(floor(v_st * float(u_frequency * 2)), 2.0);
   float   delta = abs(texcoord.x - texcoord.y);

   gl_FragColor = mix(u_color1, u_color0, delta);
```

}

图 13-11 是例 13-17 使用 u_frequency = 10, u_color0 设定为黑和 u_color1 为白的结果。

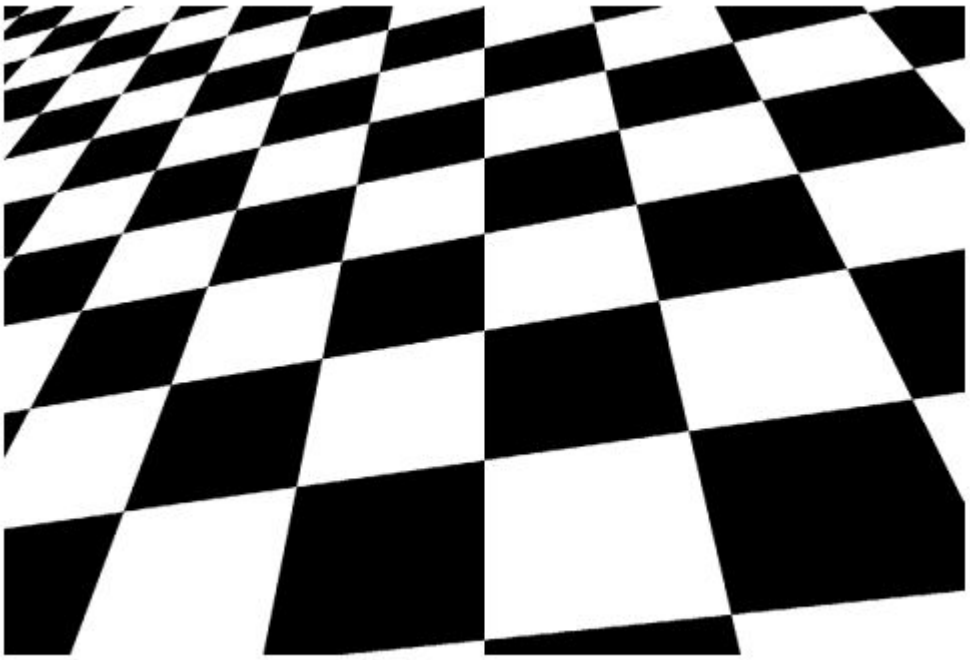


Figure 13-11 Checkerboard Procedural Texture

你看到这是容易实现的。我们看到一些锯齿，这是不能接受的。使用纹理图像检查，使用材质转换、应用较好的三线性或非线性过滤，走样被克服。我们看到一个怎样渲染抗锯齿的西洋棋图像。为平滑程序纹理，我们需要 GL_OES_standard_derivatives 扩展的内置函数，看附录 B 扩展的内置函数详解。

抗锯齿程序纹理

先进的渲染器：为移动图片创建 CGI，Anthony Apodaca 和 Larry Gritz 给出了完整的解释，怎样实现抗锯齿的程序纹理。我们使用本书描述的抗锯齿检查片段着色器。例 13-18 描述抗锯齿检查片段着色器，看代码 Chapter13/ProceduralTextures 的 CheckerAA.rfx。

例 13-18 抗锯齿检查片段着色器

```
#ifdef GL_ES
precision highp float;
#extension GL_OES_standard_derivatives : enable
#endif
uniform int     u_frequency;
uniform vec4    u_color0;
uniform vec4    u_color1;
varying vec2    v_st;
void
main()
{
   vec4    color;
   vec2    st_width;
```

```
    vec2    fuzz;
    vec2    check_pos;
    float   fuzz_max;

    // calculate the filter width.
    st_width = fwidth(v_st);
    fuzz = st_width * float(u_frequency) * 2.0;
    fuzz_max = max(fuzz.s, fuzz.t);

    // get the place in the pattern where we are sampling
    check_pos = fract(v_st * float(u_frequency));

    if (fuzz_max <= 0.5)
    {
       // if the filter width is small enough, compute the pattern
       // color by performing a smooth interpolation between the
       // computed color and the average color.
       vec2    p = smoothstep(vec2(0.5), fuzz + vec2(0.5), check_pos)
          + (1.0 - smoothstep(vec2(0.0), fuzz, check_pos));

       color = mix(u_color0, u_color1,
                   p.x * p.y + (1.0 - p.x) * (1.0 - p.y));
       color = mix(color, (u_color0 + u_color1)/2.0,
                           smoothstep(0.125, 0.5, fuzz_max));
    }
    else
    {
       // filter is too wide. just use the average color.
       color = (u_color0 + u_color1)/2.0;
    }

    gl_FragColor = color;
}
```

图 13-12 显示例 13-18 代码使用 u_frequency = 10, u_color0 设定黑, u_color1 设定为白的结果。

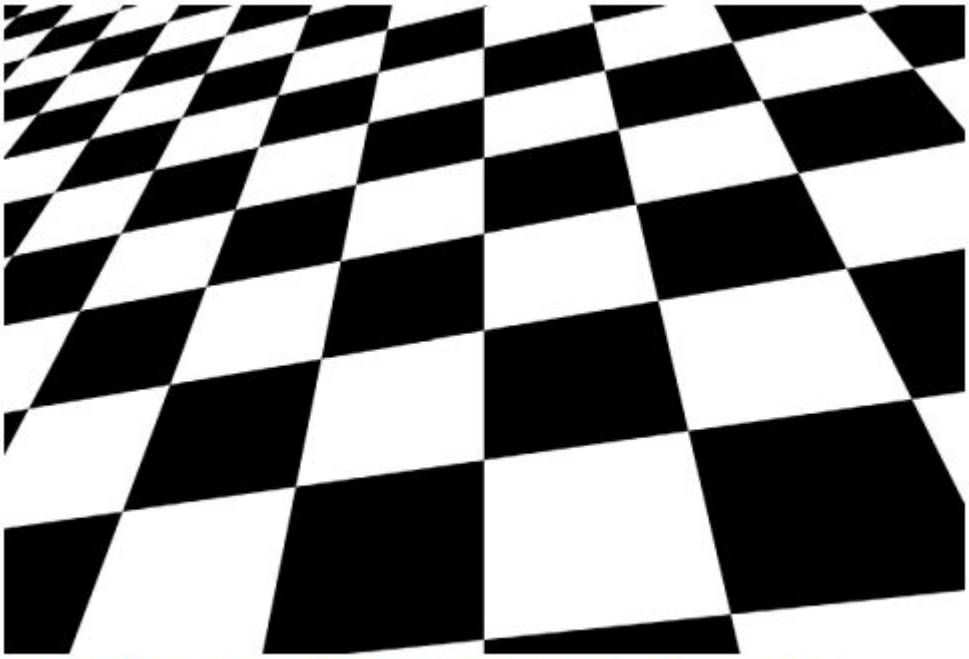


Figure 13-12 Antialiased Checkerboard Procedural Texture

为了让编程的棋盘纹理有抗锯齿效果，我们需要估算以像素为单位的纹理覆盖面积的平均值。给出代替程序纹理的函数 g(v)。需要估算以像素为单位的 g(v)覆盖区域的平均值。为这个目标，我们需要知道 g(v)的变化率，扩展 GL_OES_standard_derivatives 允许我们使用 dFdx 和 dFdy 计算 g(v) 的 x 和 y 变化率，变化率也称梯度矢量被给[dFdx(g(v)), dFdy(g(v))]。计算公式是 sqrt((dFdx(g(v))2 + dFdx(g(v))2)。也使用 abs(dFdx(g(v))) + abs(dFdy(g(v)))估计。函数 fwidth 被用于计算梯度矢量的量。如果 g(v)是标量扩展，这种方法效果很好，如果 g(v)是个点，我们需要去计算 dFdx(g(v))和 dFdy(g(v))的交叉点。在棋盘的例子中，我们需要计算 v_st.x 和 v_st.y 标量，fwidth 被用于计算 v_st.x 和 v_st.y 的过滤宽度。

使用 fwidth 计算过滤宽。我们需要知道另外的两个事情。

最小的过滤宽度 k，程序纹理不能显示任何小于 k/2 的抗锯齿效果。

程序纹理平均值超过的最大宽度。

如果 w < k/2，我们看不到任何抗锯齿效果。如果 w > k/2（或者过滤宽太大）抗锯齿发生作用。我们在例子中使用平均值。对别的 w，使用 smoothstep 平滑真实的行为和平均值。

希望提供好的解释对怎样使用程序纹理，怎样解决使用程序纹理产生的锯齿。程序纹理对很多应用来说是个大的话题。下面的参考提供一个好的开始，如果你有兴趣读更多程序纹理产生。

程序纹理更多阅读

1.Anthony A. Apodaca and Larry Gritz. Advanced Renderman: Creating CGI for Motion Pictures (Morgan Kaufmann, 1999).

2.David S. Ebert, F. Kenton Musgrave, Darwyn Peachey, Ken Perlin, and Steven Worley. Texturing and Modeling: A Procedural Approach, 3rd ed. (Morgan Kaufmann, 2002).

3.K. Perlin. An image synthesizer. Computer Graphics (SIGGRAPH 1985 Proceedings, pp. 287C296, July 1985)

4.K. Perlin. Improving noise. Computer Graphics (SIGGRAPH 2002 Proceedings, pp. 681C682)
5.K. Perlin. Making Noise. www.noisemachine.com/talk1/.
6.Pixar. The Renderman Interface Specification, Version 3.2. July 2000. renderman.pixar.com/products/rispec/index.htm.
7.Randi J. Rost. OpenGL Shading Language, 2nd ed. (Addison-Wesley Professional, 2006).

状态查询

OpenGL ES 2.0 依靠收集大量的数据和对象渲染你想要的物体。你需要编译和链接着色器项目，初始化顶点矩阵、绑定属性、指定 uniform 变量、可能装载和绑定纹理，但这仅仅是皮毛。

这有大量的 OpenGL ES 2.0 内部的操作。你能需要决定多大的视窗需要支持，或者最多多少纹理单元。所有这些值需要你的程序查询。

本章描述你的应用能使用的 OpenGL ES 2.0 变量，还有你能查询到的参数。

OpenGL ES 2.0 的字符串查询

OpenGL ES 2.0 最基本的查询像版本、可用的扩展。使用 glGetString 会以 ASCII 字符串方式返回结果。

const GLubyte* glGetString(GLenum name)

name　指定返回的参数，可以使下面中的一个：
GL_VENDOR, GL_RENDERER, GL_VERSION,
GL_SHADING_LANGUAGE_VERSION, or GL_EXTENSIONS

GL_VENDOR 和 GL_RENDERER 查询供人消费，它们没有设定格式，无论是否有用都会被初始化。

GL_VERSION 查询返回版本，格式为：

OpenGL ES <version> <vendor-specific information>

<version>是主版本（比如 2.0），跟着是副版本和可选的时期和最终释放时间（一般值厂商驱动版本值）

GL_SHADING_LANGUAGE_VERSION 查询将返回一个以 OpenGL ES GLSL ES 1.00 开始的字符串。这个字符串包含着厂商信息格式是：

OpenGL ES GLSL ES <version> <vendor-specific information>

当 OpenGL ES 版本更新到下一个版本时，版本信息将相应的改变。

最后，GL_EXTENSIONS 查询将返回相互隔开的列表，所有支持的扩展，如果返回 NULL 字符串，表示不支持扩展。

查询编译限制

很多渲染参数依靠 OpenGL ES 编译工具的能力。例如着色器有多少个纹理单元，或者最大的纹理映射尺寸或锯齿点。这些类型查询使用下面的函数。

void glGetBooleanv(GLenum pname, GLboolean* params)

void glGetFloatv(GLenum pname, GLfloat* params)

void glGetIntegerv(GLenum pname, GLint* params)

pname　指定被查询的参数

params　指定包含各种独立数据类型的、有足够空间的数组来保存相关参数的返回值

能用于查询的参数看表 14-1

表 14-1 独立的状态查询

State Variable	Description	Minimum/Initial Value	Get Function
GL_VIEWPORT	Current size of the Viewport		glGetIntegerv
GL_DEPTH_RANGE	Current depth range Values	(0, 1)	glGetFloatv
GL_LINE_WIDTH	Current line width	1.0	glGetFloatv
GL_CULL_FACE_MODE	Current face mode for polygon culling	GL_BACK	glGetIntegerv

GL_FRONT_FACE	Current vertex ordering for specifying a front-facing polygon	GL_CCW	glGetIntegerv
GL_POLYGON_OFFSET_FACTOR	Current polygon offset factor value	0	glGetFloatv
GL_POLYGON_OFFSET_UNITS	Current polygon offset units value	0	glGetFloatv
GL_SAMPLE_COVERAGE_VALUE	Current multisample coverage value	1	glGetFloatv
GL_SAMPLE_COVERAGE_INVERT	Current value of multisample inversion flag	GL_FALSE	glGetBooleanv
GL_COLOR_WRITEMASK	Current color buffer writemask value	GL_TRUE	glGetBooleanv
GL_DEPTH_WRITEMASK	Current depth buffer writemask value	GL_TRUE	glGetBooleanv
GL_STENCIL_WRITEMASK	Current stencil buffer writemask value	1s	glGetIntegerv
GL_STENCIL_BACK_WRITEMASK	Current back stencil buffer writemask value	1s	glGetIntegerv
GL_COLOR_CLEAR_VALUE	Current color buffer clear value	(0, 0, 0, 0)	glGetFloatv
GL_DEPTH_CLEAR_VALUE	Current depth buffer clear value	1	glGetIntegerv
GL_STENCIL_CLEAR_VALUE	Current stencil buffer clear value	0	glGetIntegerv
GL_SUBPIXEL_BITS	Number of subpixel bits supported	4	glGetIntegerv
GL_MAX_TEXTURE_SIZE	Maximum size of a texture	64	glGetIntegerv
GL_MAX_CUB_MAP_TEXTURE_SIZE	Maximum dimension	16	glGetIntegerv

	of a cubemap texture		
GL_MAX_VIEWPORT_DIMS			
	Dimensions of the maximum supported viewport size		glGetIntegerv
GL_ALIASED_POINT_SIZE_RANGE			
	Range of aliased point sizes	1, 1	glGetFloatv
GL_ALIASED_LINE_WIDTH_RANGE			
	Range of aliased line width sizes	1, 1	glGetFloatv
GL_NUM_COMPRESSED_TEXTURE_ FORMATS			
	Number of compressed texture formats supported		glGetIntegerv
GL_COMPRESSED_TEXTURE_FORMATS			
	Compressed texture formats supported		glGetIntegerv
GL_RED_BITS	Number of red bits in current color buffer		glGetIntegerv
GL_GREEN_BITS	Number of green bits in current color buffer		glGetIntegerv
GL_BLUE_BITS	Number of blue bits in current color buffer		glGetIntegerv
GL_ALPHA_BITS	Number of alpha bits in current color buffer		glGetIntegerv
GL_DEPTH_BITS	Number of bits in the current depth buffer	16	glGetIntegerv
GL_STENCIL_BITS	Number of stencil bits in current stencil buffer	8	glGetIntegerv
GL_IMPLEMENTATION_READ_TYPE			
	Data type for pixel components for pixel read operations		glGetIntegerv
GL_IMPLEMENTATION_READ_FORMAT			
	Pixel format for pixel read operations		

OpenGL ES 状态查询

有很多参数你能够应用修改影响 OpenGL ES 运行。当需要修改时，应用程序经常是非常有效的去追踪和修改它们的值。你能检索表 14-2 中所列出当前绑定内容的值，对每个符号，OpenGL ES 提供获取函数。

表 14-2 可修改的 OpenGL ES 状态查询

State Variable	Description	Minimum/Initial Value	Get Function
GL_ARRAY_BUFFER_BINDING			
	Currently bound	0	glGetIntegerv

	vertex attribute array binding		
GL_ELEMENT_ARRAY_BUFFER_BINDING			
	Currently bound element array binding	0	glGetIntegerv
GL_CULL_FACE_MODE	Current face culling mode	GL_BACK	glGetIntegerv
GL_FRONT_FACE	Current front-facing vertex winding mode	GL_CCW	glGetIntegerv
GL_SAMPLE_COVERAGE_VALUE			
	Current value specified for multisampling sample coverage value	1	glGetFloatv
GL_SAMPLE_COVERAGE_INVERT			
	Current multisampling coverage value inversion setting	GL_FALSE	glGetBooleanv
GL_TEXTURE_BINDING_2D			
	Current 2D texture binding	0	glGetIntegerv
GL_TEXTURE_BINDING_CUBE_MAP			
	Current cubemap texture binding	0	glGetIntegerv
GL_ACTIVE_TEXTURE	Current texture unit	0	glGetIntegerv
GL_COLOR_WRITEMASK	Color buffer writable	GL_TRUE	glGetBooleanv
GL_DEPTH_WRITEMASK	Depth buffer writable	GL_TRUE	glGetBooleanv
GL_STENCIL_WRITEMASK	Current write mask for front-facing stencil buffer	1	glGetIntegerv
GL_STENCIL_BACK_WRITEMASK			
	Current write mask or back-facing stencil buffer	1	glGetIntegerv
GL_COLOR_CLEAR_VALUE			
	Current color buffer clear value	0, 0, 0, 0	glGetFloatv
GL_DEPTH_CLEAR_VALUE	Current depth buffer clear value	1	glGetIntegerv
GL_STENCIL_CLEAR_VALUE	Current stencil buffer clear value	0	glGetIntegerv
GL_SCISSOR_BOX	Current offset and dimensions of the	0, 0, w, h	glGetIntegerv

	scissor box		
GL_STENCIL_FUNC	Current stencil test operator function	GL_ALWAYS	glGetIntegerv
GL_STENCIL_VALUE_MASK	Current stencil test value mask	1s	glGetIntegerv
GL_STENCIL_REF	Current stencil test reference value	0	glGetIntegerv
GL_STENCIL_FAIL	Current operation for stencil test failure	GL_KEEP	glGetIntegerv
GL_STENCIL_PASS_DEPTH_FAIL	Current operation for when the stencil test passes, but the depth test fails	GL_KEEP	glGetIntegerv
GL_STENCIL_PASS_DEPTH_PASS	Current operation when both the stencil and depth tests pass	GL_KEEP	glGetIntegerv
GL_STENCIL_BACK_FUNC	Current back-facing stencil test operator function	GL_ALWAYS	glGetIntegerv
GL_STENCIL_BACK_VALUE_MASK	Current back-facing stencil test value mask	1s	glGetIntegerv
GL_STENCIL_BACK_REF	Current back-facing stencil test reference value	0	glGetIntegerv
GL_STENCIL_BACK_FAIL	Current operation for back-facing stencil test failure	GL_KEEP	glGetIntegerv
GL_STENCIL_BACK_PASS_DEPTH_FAIL	Current operation for when the back-facing stencil test passes, but the depth test fails	GL_KEEP	glGetIntegerv
GL_STENCIL_BACK_PASS_DEPTH_PASS	Current operation when both the back-facing stencil and depth tests pass	GL_KEEP	glGetIntegerv
GL_DEPTH_FUNC	Current depth test comparison function	GL_LESS	glGetIntegerv
GL_BLEND_SRC_RGB	Current source RGB	GL_ONE	glGetIntegerv

	blending coefficient		
GL_BLEND_SRC_ALPHA	Current source alpha blending coefficient	GL_ONE	glGetIntegerv
GL_BLEND_DST_RGB	Current destination RGB blending coefficient	GL_ZERO	glGetIntegerv
GL_BLEND_DST_ALPHA	Current destination alpha blending coefficient	GL_ZERO	glGetIntegerv
GL_BLEND_EQUATION	Current blend equation operator	GL_FUNC_ADD	glGetIntegerv
GL_BLEND_EQUATION_RGB	Current RGB blend equation operator	GL_FUNC_ADD	glGetIntegerv
GL_BLEND_EQUATION_ALPHA	Current alpha blend equation operator	GL_FUNC_ADD	glGetIntegerv
GL_BLEND_COLOR	Current blend color	0, 0, 0, 0	glGetFloatv
GL_UNPACK_ALIGNMENT	Current byte-boundary alignment for pixel unpacking	4	glGetIntegerv
GL_PACK_ALIGNMENT	Current byte-boundary alignment for pixel packing	4	glGetIntegerv
GL_CURRENT_PROGRAM	Currently bound shader program	0	glGetIntegerv
GL_RENDERBUFFER_BINDING	Currently bound renderbuffer	0	glGetIntegerv
GL_FRAMEBUFFER_BINDING	Currently bound Framebuffer	0	glGetIntegerv

提示

OpenGL ES 2.0 使用提示去修改操作特性，允许性能和质量的偏离，你能使用下面的函数指定一个参考。

void glHint(GLenum target, GLenum mode)

target　指定设定提示，必须是 GL_GENERATE_MIPMAP_HINT 或 L_FRAGMENT_SHADER_DERIVATIVE_HINT_OES

mode　指定应该使用的执行模式的特征，有效值 GL_FASTEST 指定详细性能, GL_NICEST 指定优先的质量,或者 GL_DONT_CARE 复位任何参数选择到默认值。

使用合适的提示枚举值，任何暗示的当前能够通过 glGetIntegerv 函数恢复。

个体名字查询

OpenGL ES 2.0 提供很多你定义的个体名字—纹理、着色器、程序、顶点缓冲区、帧缓冲区、渲染缓冲区。通过调用下面的函数你能判断名字是否是当前在使用的（有效个体）。

GLboolean glIsTexture(GLuint texture)

GLboolean glIsShader(GLuint shader)

GLboolean glIsProgram(GLuint program)

GLboolean glIsBuffer(GLuint buffer)

GLboolean glIsRenderbuffer(GLuint renderbuffer)

GLboolean glIsFramebuffer(GLunit framebuffer)

texture, shader, program, buffer, renderbuffer, framebuffer　如果名字被使用，指定使用哪个独立的实体名字

非程序操作控制和查询

OpenGL ES 2.0 光栅化阶段有一些函数，像混合和背面剪切，根据你的需要被打开和关断，控制的函数如下：

void glEnable(GLenum capability)

capability　指定应该打开的特征，影响所有的渲染，直到特征被关闭

void glDisable(GLenum capability)

capability　指定应该关闭的特征

另外，你能觉得一个特性是否被使用。

GLboolean glIsEnabled(GLenum capability)

capability　如果使能了，指定是什么特征。

能够被 glEnable 和 glDisable 使用的内容看表 14-3

表 14-3 能够被 glEnable 和 glDisable 使用的内容

性能	描述
GL_CULL_FACE	丢弃多边形是那些顶点环绕顺序和指定正向模式相反(GL_CW 或 GL_CCW, 被 glFrontFace 指定)
GL_POLYGON_OFFSET_FILL	片段深度偏置值，帮助渲染的共面几何体
GL_SCISSOR_TEST	渲染时对剪切测试设定更多限制
GL_SAMPLE_COVERAGE	在多重采样操作时使用片段计算覆盖值
GL_SAMPLE_COVERAGE_TO_ALPHA	在多重采样操作时，使用片段透明度作为覆盖值
GL_STENCIL_TEST	使能模板测试
GL_DEPTH_TEST	使能深度测试
GL_BLEND	使能混合
GL_DITHER	使能抖动

着色器和程序状态查询

OpenGL ES 2.0 着色器和程序有大量的可以配置和查询的信息、属性和 uniform 变量。有一些函数提供查询着色器关联的状态。决定着色器是否关联到项目，调用下面的函数。

void glGetAttachedShaders(GLuint program, GLsizei maxcount,
GLsizei *count, GLuint *shaders)

program　查询指定程序确定附着的着色器

maxcount　应该被返回的最大着色器数目

count　实际返回的着色器数目

shaders　返回存储着色器名字的最大长度的矩阵

检索着色器源码使用：

void glGetShaderSource(GLuint shader, GLsizei bufsize,
GLsizei *length, GLchar *source)

shader　指定查询的着色器

bufsize　返回着色器源码中矩阵源码可用的数目，以位为单位。

length　返回着色器字符的长度

source　指定存储着着色器源码的 Glchars 矩阵

检索关联到着色器项目的 uniform 变量使用：

void glGetUniformfv(GLuint program, GLint location,
GLfloat *params)
void glGetUniformiv(GLuint program, GLint location,
GLint *params)

program	the program to query to retrieve the uniforms valu
location	the uniform location associated with program to retrieve the values for
params	an array of the appropriate type for storing the uniform variables values. The associated type of the uniform in the
shader	determines the number of values returned

查询 OpenGL ES 2.0 着色器语言的范围和精度使用：
void glGetShaderPrecisionFormat(GLenum shaderType,
GLenum precisionType,
GLint *range,
GLint *precision)

shaderType	specifies the type of shader, and must be either GL_VERTEX_SHADER or GL_FRAGMENT_SHADER
precisionType	specifies the precision qualifier type, and must be one ofGL_LOW_FLOAT, GL_MEDIUM_FLOAT, GL_HIGH_FLOAT, GL_LOW_INT, GL_MEDIUM_INT, or GL_HIGH_INT
range	is a two-element array that returns the minimum and maximum value for precisionType as a log base-2 number
precision	returns the precision for precisionType as a log base-2 value

顶点属性查询

顶点属性状态能被当前 OpenGL ES 2.0 环境上下文检索。使用索引检索当前顶点属性使用：void glGetVertexAttribPointerv(GLuint index, GLenum pname,
GLvoid** pointer)

index	specifies index of the generic vertex attribute array
pname	specifies the parameter to be retrieved, and must be GL_VERTEX_ATTRIB_ARRAY_POINTER
pointer	returns the address of the specified vertex attribute array

顶点属性矩阵数据元素的状态，像数据类型，能获取使用：
void glGetVertexAttribfv(GLuint index, GLenum pname,
GLfloat* params)
void glGetVertexAttribiv(GLuint index, GLenum pname,
GLint* params)

index	specifies the index of generic vertex attribute array
pname	specifies the parameter to be retrieved, and must be one of GL_VERTEX_ATTRIB_ARRAY_BUFFER_BINDING, GL_VERTEX_ATTRIB_ARRAY_ENABLED, GL_VERTEX_ATTRIB_ARRAY_SIZE, GL_VERTEX_ATTRIB_ARRAY_STRIDE, GL_VERTEX_ATTRIB_ARRAY_TYPE, GL_VERTEX_ATTRIB_ARRAY_NORMALIZED, or

GL_CURRENT_VERTEX_ATTRIB
GL_CURRENT_VERTEX_ATTRIB
returns the current vertex attribute as specified by glEnableVertexAttribeArray, and the other parameters are values specified when the vertex attribute pointer is specified by calling glVertexAttribPointer

params specifies an array of the appropriate type for storing the returned parameter values

纹理状态查询

OpenGL ES 2.0 纹理大学存储着纹理图像数据，以及图像纹理应该如何被采样的设置。纹理滤波状态，包括最小和最大的纹理过滤和纹理坐标的包装模式能使用当前绑定的纹理对象查询，下面的函数检索纹理滤波设置。

void glGetTexParameterfv(GLenum target, GLenum pname,
GLfloat* params)
void glGetTexParameteriv(GLenum target, GLenum pname,
GLint* params)

target specifies the texture target, and can either be GL_TEXTURE_2D or GL_TEXTURE_CUBE_MAP

pname specifies the texture filter parameter to be retrieved, and may beGL_TEXTURE_MINIFICATION_FILTER, GL_TEXTURE_MAGNIFICATION_FILTER, GL_TEXTURE_WRAP_S, Or GL_TEXTURE_WRAP_T

params specifies an array of the appropriate type for storing the returned parameter values

顶点缓冲区查询

顶点缓冲区对象的状态和缓冲区的使用，能被检索：

void glGetBufferParameteriv(GLenum target, GLenum pname,
GLint* params)

target specifies the buffer of the currently bound vertex buffer, and must be one of GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER

pname specifies the buffer parameter to be retrieved, and must be one ofGL_BUFFER_SIZE, GL_BUFFER_USAGE, GL_BUFFER_ACCESS, orGL_BUFFER_MAPPED

params specifies an integer array for storing the returned parameter values

另外你能检索映射缓冲区的当前地址。

void glGetBufferPointervOES(GLenum target, GLenum pname,
void** params)

target specifies the buffer of the currently bound vertex buffer, and must be one of GL_ARRAY_BUFFER or GL_ELEMENT_ARRAY_BUFFER

pname specifies the parameter to retrieve, which must be GL_BUFFER_MAP_POINTER_OES

Params specifies a pointer for storing the returned address

渲染缓冲区和帧缓冲区状态查询

渲染缓冲区的分配特性能被检索：

void glGetRenderbufferParameteriv(GLenum target,
GLenum pname,
GLint* params)

target specifies the target for the currently bound renderbuffer, and must be GL_RENDERBUFFER

pname specifies the renderbuffer parameter to retrieve, and must be one of GL_RENDERBUFFER_WIDTH, GL_RENDERBUFFER_HEIGHT, GL_RENDERBUFFER_INTERNAL_FORMAT, GL_RENDERBUFFER_RED_SIZE,GL_RENDERBUFFER_GREEN_SIZE, GL_RENDERBUFFER_BLUE_SIZE, GL_RENDERBUFFER_ALPHA_SIZE, GL_RENDERBUFFER_DEPTH_SIZE, GL_RENDERBUFFER_STENCIL_SIZE

params specifies an integer array for storing the returned parameter values

同样，当前附着的帧缓冲区能被检索：

void glGetFramebufferAttachmentParameteriv(GLenum target,
GLenum attachment, GLenum pname, GLint* params)

target specifies the framebuffer target, and must be set to GL_FRAMEBUFFER

attachment specifies which attachement point to query, and must be one of GL_COLOR_ATTACHMENT0, GL_DEPTH_ATTACHMENT or GL_STENCIL_ATTACHMENT.

pname specifies GL_FRAMEBUFFER_ATTACHMENT_OBJECT_TYPE, GL_FRAMEBUFFER_ATTACHMENT_OBJECT_NAME, GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_LEVEL, GL_FRAMEBUFFER_ATTACHMENT_TEXTURE_CUBE_MAP_FACE

Params specifies an integer array for storing the returned parameter values

OpenGL ES 和 EGL 的硬件平台

现在你应该熟悉 OpenGL ES 2.0 和 EGL1.3 的细节了。最后的章节，我们把我们的目光从 APIs 细节移开，谈谈实际的 OpenGL ES 2.0 和 EGL 编程。当开发 OpenGL ES 2.0 程序时，市场上有很多不同的硬件平台。通过讨论这些硬件平台我们讨论下面这些问题：

C++的可移植性

OpenKODE

与平台有关的着色器二进制码

目标扩展

硬件平台预览

OpenGL ES 2.0 和 EGL 1.3 对游戏和应用平台很挑剔。在多样性的平台间拷贝工作将是一个巨大的挑战。对于手持设备，一个最大的话题是设备和环境处理碎片的能力。让我们看一下现在的一些最大应用平台。

诺基亚—塞班 60 系列

高通—BREW

微软—Windows Mobil

嵌入式 linux

索尼爱立信—塞班的 UIQ

除了操作系统，还有各种使用的 CPU。大部分是基于 ARM 家族的实现，有各种特点。有些 CPU 支持原始的浮点运算，有的不支持。ARM 的处理器需要你检查校准数据的 32 位边界，可能提供给你快速浮点运算库。（或使用浮点运算）

一些操作系统—Windows Mobil 和嵌入式 linux—提供简单和熟悉的编译环境。例如微软的嵌入式 VC++包含着很多桌面版函数。另外一些 Win32 API 的子集，这让移植更容易。

其它的操作系统—例如塞班和 BREW—和 PC 平台和控制台开发非常不同。需要非常仔细的使用 C++的特性，不能使用静态全局变量，内存管理等其它方面。静态全局变量对其他平台可以使用动态连接库存储、静态只读存储器存储等，另外对代码命名，塞班提供基于 Eclipse IDE 的完整的工具链。这意味着必须学习新的调试器、项目文件（塞班叫 MMP 文件）和操作系统的 API。

在线资源

因为很多硬件平台，对不同的项目提供快速开始的向导信息是有帮助的。如果你的平台是诺基亚，看 http://forum.nokia.com 网站。那你能下载塞班的开发工具和说明文档。诺基亚提供 OpenGL ES 1.1 的 SDK，你能得到一些例子，怎么开始开发 OpenGL ES 2.0 应用的主意。索尼爱立信的设备开发使用 UIQ，在 http://developer.sonyericsson.com 网站你能找到它的信息。同样，高通的 BREW SDK 在 http://brew.qualcomm.com 网站提供 OpenGL ES 1.0 和它的扩展。写这本书时，OpenGL ES 2.0 仍然没有支持，然而使用塞班的 60 SDK 能开发 BREW 的应用，学习平台移植。高通已经宣布计划 MSM7850 使用 LT 的绘图核心支持 OpenGL ES 2.0。

开发微软的 Windows Mobile，到微软主页 http://msdn2.microsoft.com/en-us/windowsmobile/default.aspx。如果你已经有 Microsoft Visual Studio 2005,你能下载 Windows Mobile 6 SDK，在 VS2005 上开发。大量设备支持 Windows Mobile 像 Moto Q, Palm Treo 750, Pantech Duo, HTC Touch 等等。虽然还没有支持 OpenGL ES 2.0 的，但不久就会出现。

最后对嵌入式 linux，看 www.linuxdevices.com 网站。有一些嵌入式设备在 2007 年进入市场，像诺基亚 N800,如果想开发 OpenGL ES 2.0 应用，可以在 linux（不是嵌入式）下进行。模拟技术已经释放了 OpenGL ES 2.0Linux 下的包装。可在 www.powervrin-

sider.com 网站下载。

C++移植性

开发 OpenGL ES 2.0 应用第一件你需要确定的是你使用何种语言。很多平台选择普通平庸的 C，也有好的理由。C++移植性是个显著的原因，因为不同的平台对 C++特性支持的不同。C++的特性不被支持因为这对编译器是很难得事情。记着，我们工作在嵌入式平台，内存和电力消耗是便携设备操作系统重要的目标。这意味着 C++的特性不能在手持设备平台上应用。

例如，塞班 8 不支持 C++异常处理。这意味着使用异常处理的 C++代码不能在塞班 8 上移植（塞班 8 有自己的异常处理，但不是 C++的标准方法）。缺少异常处理，也意味着塞班 8 不支持标准的库（STL）。为使用 STL，需要扔掉异常处理。结果是，你能使用塞班自己的容器类。过长和过短的使用 STL 将不能在某些平台移植。

另一个塞班 8 不支持异常处理的后果是程序必须自己手工管理和清除堆暂。这就是说，C++上支持的新操作失败，必须能产生异常。塞班上程序有义务自己管理清除堆暂本身和目标被使用双相结构创建。在便携引擎上的这种代码不是一个可选项因此一些应用选择在塞班上自己管理内存。它们开始时分配需要大小的内存块，管理所有自己分配的，以至于塞班系统上清洁代码不会很小。

很多 C++上的可移植特性在塞班 9 上是固定的，这让你在选择硬件平台受到限制。很多特性是否在硬件平台上支持不确定。例如，运行时间信息、异常、多继承、STL 可能不支持。为可移植性，你应限制 C++的使用，或者直接使用 C。

如果使用 C++，下面的特性你应该尽量避免：

运行时间类型信息—例如使用 dynamic_cast，要求知道一个类的运行时间类型。

异常—标准 C++平台 try-catc 机制在一些操作系统上不支持

标准模板库—虽然提供很多有用的类，但要求平台支持异常处理

多继承—一些平台 C++编译器不允许类继承多个类。

全局数据—一些平台在 ROM 中存储应用，不支持静态可写的数据

这个列表不是完整的，但代表了硬件平台已经出现的题目。如果供应厂商提供了非标准编译工具，你可能问你自己我使用什么语言好。有用的回答是，不是宣称支持 C++不支持特性的语言。这是可能的未来支持 C++的全部特性。现在，我们必须选择已有的平台厂商，调整我们自己的代码。

OpenKODE

各个平台除了 C++不一致外，另一个移植主要的障碍是操作系统的 APIs。例如输入、输出，获取文件、时间、数学函数以及事件句柄在各个操作系统上是不同的。处理这些不同是游戏开发商的帽子戏法。很多游戏引擎会抽取不同平台的图层来编写。便携式的部分代码不会用到到任何特定于操作系统的功能调用而是使用抽象层。

在掌上电脑平台的问题，是在大量手持平台，写你自己的抽象层比较困难的，此外，在不同功能的操作系统上找到一个普遍的特性是很困难的，特别是一些新的手持平台。幸运的是，Khronos 集团认识到这个重大障碍，并发明了一种新的 API 来处理它，即所谓的 OpenKODE。

OpenKODE 1.0 规格书在 2008 年释放。OpenKODE 提供一个标准的 APIs（包括 OpenGL ES 和 EGL）让应用能够顺序获取系统的函数。OpenKODE 核心提供事件 APIs、内存分配、文件通道、输入输出、算数运算、网络接口等等。为介绍使用 OpenKODE 观念，提供一个使用 OpenKODE 的 Hello Triangle 例子。我们移除了系统特定的函数，移除了依靠 ES 应用框架的调用。

例子代码在 Chapter_15/Hello_Triangle_KD，例 15-1 显示部分源码。

例 15-1 使用 OpenKODE 的 Hello Triangle

```
#include <KD/kd.h>
#include <EGL/egl.h>
#include <GLES2/gl2.h>
typedef struct
{
   // Handle to a program object
   GLuint programObject;
   // EGL handles
   EGLDisplay eglDisplay;
   EGLContext eglContext;
   EGLSurface eglSurface;
} UserData;
///
// Create a shader object, load the shader source, and
// compile the shader.
//
GLuint LoadShader(GLenum type, const char *shaderSrc) {
   GLuint shader;
   GLint compiled;
   // Create the shader object
   shader = glCreateShader(type);
   if(shader == 0)
      return 0;
   // Load the shader source
   glShaderSource(shader, 1, &shaderSrc, NULL);
   // Compile the shader
   glCompileShader(shader);
   // Check the compile status
   glGetShaderiv(shader, GL_COMPILE_STATUS, &compiled);
   if(!compiled)
   {
      GLint infoLen = 0;
      glGetShaderiv(shader, GL_INFO_LOG_LENGTH, &infoLen);
      if(infoLen > 1)
      {
         char* infoLog = kdMalloc(sizeof(char) * infoLen);
         glGetShaderInfoLog(shader, infoLen, NULL, infoLog);
         kdLogMessage(infoLog);
         kdFree(infoLog);
      }
      glDeleteShader(shader);
      return 0;
   }
   return shader;
```

```
}
///
// Initialize the shader and program object
// int Init(UserData *userData) {
   GLbyte vShaderStr[] =
      "attribute vec4 vPosition;   \n"
      "void main()                 \n"
      "{                           \n"
      "   gl_Position = vPosition; \n"
      "}                           \n";
   GLbyte fShaderStr[] =
      "precision mediump float;\n"\
      "void main()                                  \n"
      "{                                            \n"
      "  gl_FragColor = vec4(1.0, 0.0, 0.0, 1.0); \n"
      "}                                            \n";
   GLuint vertexShader;
   GLuint fragmentShader;
   GLuint programObject;
   GLint linked;
   // Load the vertex/fragment shaders
   vertexShader = LoadShader(GL_VERTEX_SHADER, vShaderStr);
   fragmentShader = LoadShader(GL_FRAGMENT_SHADER, fShaderStr);
   // Create the program object
   programObject = glCreateProgram();
   if(programObject == 0)
      return 0;
   glAttachShader(programObject, vertexShader);
   glAttachShader(programObject, fragmentShader);
   // Bind vPosition to attribute 0
   glBindAttribLocation(programObject, 0, "vPosition");
   // Link the program
   glLinkProgram(programObject);
   // Check the link status
   glGetProgramiv(programObject, GL_LINK_STATUS, &linked);
   if(!linked)
   {
      GLint infoLen = 0;
      glGetProgramiv(programObject, GL_INFO_LOG_LENGTH, &infoLen);
      if(infoLen > 1)
      {
         char* infoLog = kdMalloc(sizeof(char) * infoLen);
         glGetProgramInfoLog(programObject, infoLen, NULL, infoLog);
         kdLogMessage(infoLog);
```

```
            kdFree(infoLog);
        }
        glDeleteProgram(programObject);
        return FALSE;
    }
    // Store the program object
    userData->programObject = programObject;
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f);
    return TRUE;
}
///
// Draw a triangle using the shader pair created in Init()
// void Draw(UserData *userData) {
    GLfloat vVertices[] = {  0.0f,  0.5f, 0.0f,
                            -0.5f, -0.5f, 0.0f,
                             0.5f, -0.5f, 0.0f };
    // Set the viewport
    glViewport(0, 0, 320, 240);
    // Clear the color buffer
    glClear(GL_COLOR_BUFFER_BIT);
    // Use the program object
    glUseProgram(userData->programObject);
    // Load the vertex data
    glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, vVertices);
    glEnableVertexAttribArray(0);
    glDrawArrays(GL_TRIANGLES, 0, 3);
    eglSwapBuffers(userData->eglDisplay, userData->eglSurface); }
///
// InitEGLContext()
//
// Initialize an EGL rendering context and all associated elements
//
EGLBoolean InitEGLContext(UserData *userData,
                          KDWindow *window,
                          EGLConfig config) {
    EGLContext context;
    EGLSurface surface;
    EGLint contextAttribs[] = { EGL_CONTEXT_CLIENT_VERSION, 2,
                                EGL_NONE, EGL_NONE };
    // Get native window handle
    EGLNativeWindowType hWnd;
    if(kdRealizeWindow(window, &hWnd) != 0)
    {
        return EGL_FALSE;
```

```
    }
    surface = eglCreateWindowSurface(userData->eglDisplay, config,
                                     hWnd, NULL);
    if(surface == EGL_NO_SURFACE)
    {
       return EGL_FALSE;
    }
    // Create a GL context
    context = eglCreateContext(userData->eglDisplay, config,
                               EGL_NO_CONTEXT, contextAttribs );
    if(context == EGL_NO_CONTEXT)
    {
       return EGL_FALSE;
    }
    // Make the context current
    if(!eglMakeCurrent(userData->eglDisplay, surface, surface,
                                context))
    {
       return EGL_FALSE;
    }
    userData->eglContext = context;
    userData->eglSurface = surface;
    return EGL_TRUE;
}
///
// kdMain()
//
// Main function for OpenKODE application
//
KDint kdMain(KDint argc, const KDchar *const *argv) {
    EGLint attribList[] =
    {
        EGL_RED_SIZE,       8,
        EGL_GREEN_SIZE,     8,
        EGL_BLUE_SIZE,      8,
        EGL_ALPHA_SIZE,     EGL_DONT_CARE,
        EGL_DEPTH_SIZE,     EGL_DONT_CARE,
        EGL_STENCIL_SIZE,   EGL_DONT_CARE,
        EGL_NONE
    };
    EGLint majorVersion,
       minorVersion;
    UserData userData;
    EGLint numConfigs;
```

```
   EGLConfig config;
   KDWindow *window = KD_NULL;
   userData.eglDisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
   // Initialize EGL
   if(!eglInitialize(userData.eglDisplay, &majorVersion,
                     &minorVersion) )
   {
      return EGL_FALSE;
   }
   // Get configs
   if(!eglGetConfigs(userData.eglDisplay, NULL, 0, &numConfigs))
   {
      return EGL_FALSE;
   }
   // Choose config
   if(!eglChooseConfig(userData.eglDisplay, attribList, &config,
      1, &numConfigs))
   {
      return EGL_FALSE;
   }
   // Use OpenKODE to create a Window
   window = kdCreateWindow(userData.eglDisplay, config, KD_NULL);
   if(!window)
      kdExit(0);
   if(!InitEGLContext(&userData, window, config))
      kdExit(0);
   if(!Init(&userData))
      kdExit(0);
   // Main Loop
   while(1)
   {
      // Wait for an event
      const KDEvent *evt = kdWaitEvent(0);
      if ( evt )
      {
         // Exit app
         if(evt->type == KD_EVENT_WINDOW_CLOSE)
            break;
      }
      // Draw frame
      Draw(&userData);
   }
   // EGL clean up
   eglMakeCurrent(0, 0, 0, 0);
```

```
    eglDestroySurface(userData.eglDisplay, userData.eglSurface);
    eglDestroyContext(userData.eglDisplay, userData.eglContext);
    // Destroy the window
    kdDestroyWindow(window);
    return 0;
}
```

OpenKODE 的核心函数使用 kd 前缀，主函数是 kdMain()。你注意到例子使用 OpenKODE 函数创建设置所有的窗口。EGL 被使用创建和初始化 OpenGL ES 渲染视口。另外，使用 OpenKODE 分配内存（kdMalloc() and kdFree()）和注册消息（kdLogMessage()）。

例子显示了使用 OpenKODE 能够不使用任何操作系统指定的回调。有很多 OpenKODE 函数。整个应用程序架构位于 OpenKODE 之上，会得到很多便利性。一个完整的 OpenKODE 内容需要一本书来讲述，我们只是将一些皮毛。如果你有兴趣使用 OpenKODE，去 www.khronos.org/ 下载规格书。

平台相关的着色器二进制码

除了代码移植外，另一个 OpenGL ES 2.0 的话题是编译和发行着色器二进制码。OpenGL ES 2.0 标准提供了一种机制，即应用程序可以提供其着色器在预编译的二进制格式。这对一些平台是非常理想的。首先，减小装载时间，因为驱动程序不需要编辑着色器。第二，离线的着色器将能够做更多优化工作，因为他们不必在同样的内存上运行，时间消耗在在线编译上。第三，一些 OpenGL ES 2.0 工具仅仅支持着色器二进制码，这意味着你必须离线编译你的着色器。

作为 OpenGL ES 2.0 开发者，你必须考虑多种平台的着色器二进制码。首先，二进制码是内置的，不能移植的。二进制码必须和设备、GPU、甚至驱动和操作系统版本紧密相连。二进制着色器格式被定义是不透明的。设备厂商是自由的以任何他们希望的非移植格式存储二进制着色器。一般，厂商会以硬件二进制码形式存储着色器。

第二个话题是没有任何 API 定义产生二进制码机制的。每个设备和 GPU 厂商是自由的建立自己的工具组件产生二进制着色器，这就需要你自己去确保你的引擎能被厂商的工具链支持。例如 AMD 提供 MakeBinaryShader.exe 窗口工具，编译代码成二进制。AMD 也提供库接口，BinaryShader.lib，它提供直接从应用程序编译执行着色器的功能。模拟技术提供称为 PVRUniSCo 的二进制编译工具链和成为 PVRUniSCo 编辑器。你也会看到其他的厂商提供他们自己的二进制编译工具。

第三个话题是厂商扩展，定义二进制着色器可能的限制。举例来书，当使用特定的 OpenGL ES 状态矢量时，着色器编译二进制可能被优化（或者仅仅是偶数功能）。
作为一个例子，一个供应商可能定义，着色器编译过程中定义的输入，只用于片段混合或多重采样，二进制可能无效如果未启用这些功能有效。这些限制是留给供应商来定义，所以鼓励应用开发人员和供应商进行协商确定一个给定的着色器的二进制格式特定的扩展。

你的引擎需要支持预处理路径，让你可以选择设备供应商的一个工具，及存储一个二进制着色器工程。AMD 的工具是可选择的一个（像其他厂商一样）。AMD 将提供了一个扩展，使应用程序可以调用到驱动程序检索编译的程序的二进制码。换句话数，程序能在驱动中在线装载和编译着色器，要求返回最后的二进制码，存储到文件中。得到的二进制码和离线编译工具编译的格式是一样的。这种方法的先进性是应用程序能够使用驱动编译运行着色器（或者在安装时），然后读回着色器二进制码，直接存储到设备。你应该在你的引擎中准备好掌握这种二进制着色器编译方法。

目标扩展

这本书，我们已经介绍给你很多扩展像 3D 纹理、ETC、深度纹理和其它衍生物。本书中

的扩展是 Khronos 组织批准的，能被很多厂商支持，但不是所有的厂商支持这些扩展。他们不是标准的一部分。例如 AMD 和 Imagination 技术提供它们自己的硬件支持的纹理压缩格式的扩展。为挖掘硬件的潜力，要学会使用这些扩展。

为保持不同 OpenGL ES 设备的可移植性，在你的程序里必须有合适的后备处理。如果你使用扩展，你必须检查扩展字符串是否包含在 GL_EXTENSIONS 中。使用扩展扩充着色器语言，你必须使用#extension 机制确保使能它们。如果没有扩展可用，你必须写后备处理方法。例如你使用里一些压缩纹理格式，但如果扩展不可用，你必须保持装载非压缩纹理格式的能力。如果你使用 3D 扩展，你需要提供后备的纹理和着色器处理替代这个效果。

如果你想保证跨平台的移植性，而又不想为特定的扩展留后备方法，不要去使用它们。鉴于已经存在于不同操作系统和设备的能力差异，你又不想让这个问题变得更糟，你使用 OpenGL ES 2.0 写的程序可以不考虑移植性。好的消息是写便于移植的 OpenGL ES 2.0 程序对比桌面 OpenGL 没有更难。检查的扩展和写作备用路径，是一个不争的事实，如果你想利用最新的 OpenGL ES 2.0 的实现提供的功能，你需要这样做。（终于混弄完了）