

从零开始学习 OpenGL ES

注:

英文原版地址:

<http://iphonedevdevelopment.blogspot.com/>

一. 基本概念

OpenGL 数据类型

首先我们要讨论的是 OpenGL 的数据类型。因为 OpenGL 是一个跨平台的 API，数据类型的大小会随使用的编程语言以及处理器（64 位，32 位，16 位）等的不同而不同，所以 OpenGL 定义了自己的数据类型。当传递数据到 OpenGL 时，你应该坚持使用这些 OpenGL 的数据类型，从而保证传递数据的尺寸和精度正确。不这样做的后果是可能会导致无法预料的结果或由于运行时的数据转换造成效率低下。不论平台或语言实现的 OpenGL 都采用这种方式定义数据类型以保证在各平台上数据的尺寸一致，并使平台间 OpenGL 代码移植更为容易。

下面是 OpenGL 的各种数据类型:

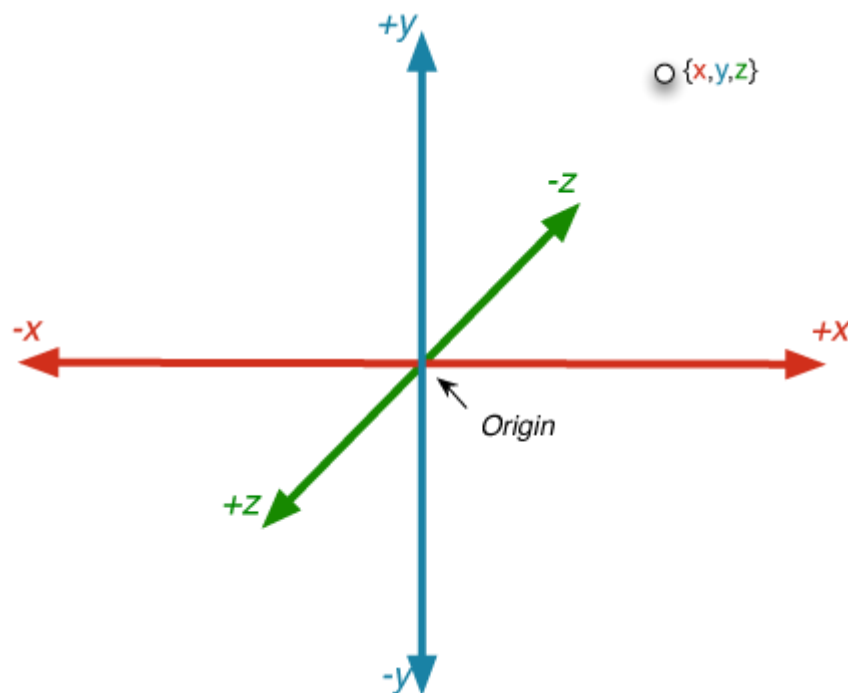
- **GLenum:** 用于 GL 枚举的无符号整型。通常用于通知 OpenGL 由指针传递的存储于数组中数据的类型（例如，GL_FLOAT 用于指示数组由 GLfloat 组成）。
- **GLboolean:** 用于单布尔值。OpenGL ES 还定义了自己的“真”和“假”值（GL_TRUE 和 GL_FALSE）以避免平台和语言的差别。当向 OpenGL 传递布尔值时，请使用这些值而不是使用 YES 或 NO（尽管由于它们的定义实际没有区别，即使你不小心使用了 YES 或 NO。但是，使用 GL-定义值是一个好的习惯。）
- **GLbitfield:** 用于将多个布尔值（最多 32 个）打包到单个使用位操作变量的四字节整型。我们将在第一次使用位域变量时详细介绍，请参阅 wikipedia
- **GLbyte:** 有符号单字节整型，包含数值从 -128 到 127
- **GLshort:** 有符号双字节整型，包含数值从 -32,768 到 32,767
- **GLint:** 有符号四字节整型，包含数值从 -2,147,483,648 到 2,147,483,647
- **GLsizei:** 有符号四字节整型，用于代表数据的尺寸（字节），类似于 C 中的 `size_t`
- **GLubyte:** 无符号单字节整型，包含数值从 0 到 255。
- **GLushort:** 无符号双字节整型，包含数值从 0 到 65,535
- **GLuint:** 无符号四字节整型，包含数值从 0 到 4,294,967,295
- **GLfloat:** 四字节精度 IEEE 754-1985 浮点数
- **GLclampf:** 这也是四字节精度浮点数，但 OpenGL 使用 GLclampf 特别表示数值为 0.0 到 1.0
- **GLvoid:** void 值用于指示一个函数没有返回值，或没有参数
- **GLfixed:** [定点数](#) 使用整数存储实数。由于大部分计算机处理器在处理整数比处理浮点数快很多，这通常是对 3D 系统的优化方式。但因为 iPhone 具有用于浮点运算的矢量处理器，我们将不讨论定点运算或 **GLfixed 数据类型**。
- **GLclampx:** 另一种定点型，用于使用定点运算来表示 0.0 到 1.0 之间的实数。正如 **GLfixed**，我们不会讨论或使用它。

OpenGL ES（至少 iPhone 目前所使用的版本）不支持 8 字节（64 位）数据类型，如 long 或 double。OpenGL 其实具有这些大型数据类型，但考虑到大部分嵌入式设备屏幕尺寸以及可能为它们所写的程序类型而且使用它们有可能对性能造成不利的影响，最后的决定是在 OpenGL ES 中排除这些数据类型。

点或顶点

3D 图像的最小单位称为 **点 (point)** 或者 **顶点 vertex**。它们代表三维空间中的一个点并用来建造更复杂的物体。多边形就是由点构成，而物体是由多个多边形组成。尽管通常 OpenGL 支持多种多边形，但 OpenGL ES 只支持三角形（即三角形）。

如果你回忆高中学过的几何学，你可能会记得所谓[笛卡尔坐标](#)。基本概念是在空间中任选一点，称作**原点**。然后你可以通过参照原点并使用三个代表三维的数值指定空间中的任意一点，坐标是由三个想象的通过原点线表示的。从左至右的想象直线叫 **x**-轴。沿着 **x**-轴从左至右数值变大，向左移动数值变小。原点左方 **x** 为负值，右边为正值。另外两轴同理。沿 **y** 轴向上，**y** 值增加，向下 **y** 值减小。原点上方 **y** 为正，原点下方为负。对于 **z** 轴，当物体离开观察者，数值变小，向观察者移动（或超出观察者），数值变大。原点前方 **z** 值为正，原点之后为负。下图帮助说明了这一点：



Note: iPhone 上另一种绘图框架 Core Graphics 使用了稍微不同的坐标系，当向屏幕上方移动时 **y** 值减小，而向下移动 **y** 值增加。

沿各轴增加或减小的数值是以任意刻度进行的 – 它们不代表任何真实单位，如英尺，英寸或米等。你可以选择任何对你的程序有意义的刻度。如果你想设计的游戏以英尺为单位，你可以那样做。如果你希望单位为毫米，同样可行。OpenGL 不管它对最终用户代表什么，只是将它作为单位处理，保证它们具有相同的距离。

由于任何物体在三维空间中的方位可以由三个数值表示，物体的位置通常在 OpenGL 中由使用一个三维数组的三个 GLfloat 变量表示，数组中的第一项（索引 0）为 **x** 位置，第二项（索引 1）为 **y** 位置，第三项（索引 2）为 **z** 位置。下面是一个创建 OpenGL ES 顶点的简单例子：

```
GLfloat vertex[3];  
  
vertex[0] = 10.0;      // x  
vertex[1] = 23.75;     // y  
vertex[2] = -12.532;   // z
```

在 OpenGL ES 中，通常将场景中所有构成所有或部分物体的提交为**顶点数组**。一个顶点数组是包括场景中部分或所有顶点数据的简单数组。我将在系列的下一篇教程中讨论，有关顶点数组要记住的是它们的大小是基于呈现的顶点数乘以三（三维空间绘图）或二（二维空间绘图）。所以一个包含六个三维空间中的三角形的顶点数组由 54 个 GLfloat 组成，因为每个三角形有三个顶点，而每个顶点有三个坐标，即 $6 \times 3 \times 3 = 54$ 。

处理所有这些 **GLfloat** 是很痛苦的事情。幸运的是，有一个容易的方法。我们可以定义一个数据结构了保存多个顶点，像这样：

```
typedef struct {
    GLfloat x;
    GLfloat y;
    GLfloat z;
} Vertex3D;
```

通过这样做，我们的代码可读性更强：

```
Vertex3D vertex;
vertex.x = 10.0;
vertex.y = 23.75;
vertex.z = -12.532;
```

现在由于 Vertex3D 由三个 GLfloat 组成，向 Vertex3D 传递指针与向数组传递一个包含三个 GLfloat 的数组的指针完全一样。对于电脑而言毫无分别；两者具有同样的尺寸和同样的字节数以及 OpenGL 需要的同样的顺序。将数据分组到数据结构只是让程序员感到更容易，处理起来更方便。如果你下载了文章开头处的 Xcode 模板，你会发现此数据结构以及我后面将讨论的各种函数都定义在文件 OpenGLCommon.h 中。还有一个内联函数用于创建单个顶点：

```
static inline Vertex3D Vertex3DMake(CGFloat inX, CGFloat inY, CGFloat inZ)
{
    Vertex3D ret;
    ret.x = inX;
    ret.y = inY;
    ret.z = inZ;
    return ret;
}
```

如果你回忆起几何学（如果不记得也不要紧）的内容，你会知道空间中两点间的距离是使用下面公式计算的：

$$\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

我们可以在一个简单的内联函数中实现这个公式来计算三维空间中任何两点间的直线距离：

```
static inline GLfloat Vertex3DCalculateDistanceBetweenVertices (Vertex3D first, Vertex3D second)
{
    GLfloat deltaX = second.x - first.x;
    GLfloat deltaY = second.y - first.y;
```

```

GLfloat deltaZ = second.z - first.z;
return sqrtf(deltaX*deltaX + deltaY*deltaY + deltaZ*deltaZ);
};

```

三角形

由于 OpenGL ES 仅支持三角形，因此我们可以通过创建一个数据结构将三个顶点组合成一个三角形物体。

```

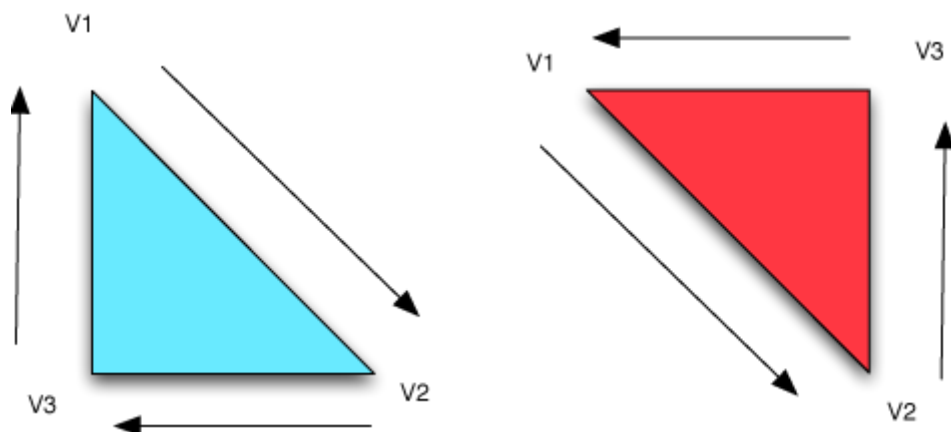
typedef struct {
    Vertex3D v1;
    Vertex3D v2;
    Vertex3D v3;
} Triangle3D;

```

一个 `Triangle3D` 实际上与一个九个 `GLfloat` 构成的数组是完全一样的，因为我们通过顶点和三角形而不是 `GLfloat` 数组来构建物体，所以它能帮助我们更容易地处理我们的代码。

然而关于三角形你需要知道更多的事情。在 OpenGL 中有一个概念叫**卷绕(winding)**，它表示顶点绘制的次序是重要的。不像真实世界中的物体，OpenGL 中的多边形通常都不会有两面。它们只有一面，被当做 **front face (前面)**，三角形只有其 front face 面对观察者时才可见。可以设置 OpenGL 将多边形作为两面处理，但默认状态下，三角形只有一个可见面。通过知道哪一个面是多边形的前面或可见面，才能使 OpenGL 只做一半的计算。

尽管有时多边形也可以独立存在，需要绘制其背面，但通常三角形是一个大物体的一部分，其面对物体内部的一面永远也不可见。不被绘制的一面称为 **backface(背面)**，OpenGL 是通过观察顶点的绘制次序来确定 front face 和 backface 的。以反时针次序绘制顶点的构成的面是 frontface (默认，可以改变)。由于 OpenGL 可以很容易确定哪个三角形对用户可见，所以它使用了一种称为 **Backface Culling (隐面消除)** 的技术来避免绘制视窗中多边形的不可见面。下一篇文章将讨论视窗，现在你可将其想象成一个虚拟摄像或观察 OpenGL 世界的虚拟窗口。



上图中，左边青色的三角形是 backface，因此将不可见。而右方的三角形是 frontface，所以将被绘制。

本系列的下一篇文章将设定一个 OpenGL 的虚拟世界并使用 `Vertex3D` 和 `Triangle3D` 进行一些基本绘图。再后，我们将讨论**变换**，它使用线性代数在虚拟世界中移动物体。

二. 简单绘图概述

还有许多理论知识需要讨论，但与其花许多时间在复杂的数学公式或难以理解的概念上，还不如让我们开始熟悉 OpenGL ES 的基本绘图功能。

请下载 [OpenGL Xcode 项目模板](#)。我们使用此模板而不是 Apple 提供的模板。你可以解压到下面目录来安装它：
/Developer/Platforms/iPhoneOS.platform/Developer/Library/Xcode/Project Templates/Application/

此模板用于全屏 OpenGL 程序，它具有一个 OpenGL 视图以及相应的视图控制器。大部分时候你不需要动到此视图。此视图用于处理一些诸如缓存切换之类的事物，但在两处调用了其控制器类。

首先，当设定视图时，调用了一次控制器。调用视图控制器的 `setupView:` 方法使控制器有机会增加所需的设定工作。这里是你设定视口，添加光源以及进行其他项目相关设定的地方。现在我们将忽略此方法。此方法中已经有非常基本的设定以允许你进行简单地绘图。

控制器的 `drawView:` 方法根据常数 `kRenderingFrequency` 的值定期地被调用。`kRenderingFrequency` 的初始值为 15.0，表示 `drawView:` 方法每秒钟被调用 15 次。如果你希望改变渲染的频率，你可以在 `ConstantsAndMacros.h` 中找到此常数的定义。

首先加入下列代码到 `GLViewController.m` 的 `drawView:` 方法中：

```
- (void)drawView:(GLView*)view;
{
    Vertex3D    vertex1 = Vertex3DMake(0.0, 1.0, -3.0);
    Vertex3D    vertex2 = Vertex3DMake(1.0, 0.0, -3.0);
    Vertex3D    vertex3 = Vertex3DMake(-1.0, 0.0, -3.0);
    Triangle3D  triangle = Triangle3DMake(vertex1, vertex2, vertex3);

    glLoadIdentity();
    glClearColor(0.7, 0.7, 0.7, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnableClientState(GL_VERTEX_ARRAY);
    glColor4f(1.0, 0.0, 0.0, 1.0);
    glVertexPointer(3, GL_FLOAT, 0, &triangle);
    glDrawArrays(GL_TRIANGLES, 0, 9);
    glDisableClientState(GL_VERTEX_ARRAY);
}
```

在讨论我们到底做了什么之前，先运行一下，你应该看到以下画面：



这是个简单的方法；如果你试过了，你可能已经知道我们到底做了什么，但这里我们还是一起过一遍。因为我们的任务是画三角形，所以需要三个顶点，因此我们创建三个[上一篇文章中讨论过的](#) `Vertex3D` 对象：

```
Vertex3D    vertex1 = Vertex3DMake(0.0, 1.0, -3.0);
Vertex3D    vertex2 = Vertex3DMake(1.0, 0.0, -3.0);
Vertex3D    vertex3 = Vertex3DMake(-1.0, 0.0, -3.0);
```

你应该注意到了三个顶点的 `z` 值是一样的，其值(-3.0)是处于原点 “之后” 的。因为我们还没有做任何改变，所以我们是站在原点上观察虚拟世界的，这是默认的起点位置。将三角形放置在 `z` 值为-3 处，可以保证我们可以在屏幕上看到它。随后，我们创建一个由这三个顶点构成的三角形。

```
Triangle3D  triangle = Triangle3DMake(vertex1, vertex2, vertex3);
```

这些代码很容易理解，对吗？但是，在幕后，电脑是将其视为一个包含 9 个 `GLfloat` 的数组。如下：

```
GLfloat  triangle[] = {0.0, 1.0, -3.0, 1.0, 0.0, -3.0, -1.0, 0.0, -3.0};
```

并不是完全相同 - 这里有一个很小但很重要的区别。在我们的示例中，我们传递给 OpenGL 的是 `Triangle3D` 对象的地址（即 `&triangle`），但在第二个使用数组的示例中，由于 C 数组是指针，我们传递的是数组。现在不需要考虑太多，因为这将是最后一次我用这种方式（第二种方法）定义一个 `Triangle3D` 对象。等一下我将解释原因，但现在让我们先过一遍代码。下一步我们做的是加载单位矩阵。我将花至少一整篇文章讨论变换矩阵。我们暂且将其视为 OpenGL 的“复位开关”。它将清除虚拟世界中的一切旋转，移动或其他变化并将观察者置于原点。

```
glLoadIdentity();
```

之后，我们告诉 OpenGL 所有的绘制工作是在一个灰色背景上进行的。OpenGL 通常需要用四个钳位值来定义颜色。上一篇文章中有提过，钳位浮点数是 0.0 到 1.0 之间的浮点数。我们通过定义红，绿，蓝以及 `alpha` 元素来定义颜色，`alpha` 值定义了颜色之后物体的透视程度。现在暂时不用管它，将其设为 1.0，代表完全不透明。

```
glClearColor(0.7, 0.7, 0.7, 1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

在 OpenGL 中要定义白色，我们需要将四个元素全部设为 1.0。要定义不透明的黑色，则定义红，绿，蓝为 0.0，alpha 为 1.0。上例代码的第二行是通知 OpenGL 清除以前的一切图形并将其设为 clear 颜色。

你可能想知道 `glClear()` 的两个参数是什么意思。简单地说，它们是存储与位域中的常量。OpenGL 保存了一系列 **缓存 (buffers)**，即用于绘图各方面的内存块。将这两个值进行逻辑或是通知 OpenGL 清除两个不同的缓存 – **颜色缓存 (color buffer)** 和 **深度缓存 (depth buffer)**。颜色缓存保存当前帧各像素的颜色。基本上就是你在屏幕上看到的。深度缓存 (有时也称为 “z-buffer”) 保存每个潜在像素离观察者距离的信息。使用此信息可以确定一个像素是否需要被绘制出来。这两个缓存是 OpenGL 中最常见的缓存。还有其他一些缓存，如模板缓存 (stencil buffer) 和累积缓存 (accumulation buffer)，但现在我们暂时不讨论这些。我们现在只需记住在绘制一帧之前，必须清除这两个缓存以保证不会和以前的内容混杂。

然后，我们要启动 OpenGL 的一项称为 **vertex arrays** (顶点数组) 的特性。此特性可能只需要 `setupView:` 方法中启动一次，但作为基本准则，我喜欢启动和禁止我使用的功能。你永远也不会知道是否另一段代码会做不同处理。如果你打开你需要的功能然后关闭它，产生问题的几率将大为减小。就本例来说，如果另一个类不使用顶点数组而使用顶点缓存的话，任何一段代码遗留了启动了的特性或没有显性启动其需要的特性，这一段或两段代码都会导致不可预知的结果。

```
glEnableClientState(GL_VERTEX_ARRAY);
```

接下来我们设置了绘图时所需的颜色。此行代码将绘图颜色设为鲜艳的红色。

```
glColor4f(1.0, 0.0, 0.0, 1.0);
```

现在，直到下次调用 `glColor4f()` 前所有的图形都是以红色绘制。有一些例外的情况，例如绘制纹理形状时，但基本上，这样设定颜色可以使颜色保持。

由于我们使用顶点数组，我们必须通知 OpenGL 顶点的数组在什么地方。记住，顶点数组只是一个 `GLfloat` 的 C 数组，每三个值代表一个顶点。我们创建了 `Triangle3D` 对象，但在内存中，它完全等同于 9 个连续的 `GLfloat`，所以我们可以传递此三角形对象的地址。

```
glVertexPointer(3, GL_FLOAT, 0, &triangle);
```

`glVertexPointer()` 的第一个参数指示了多少个 `GLfloat` 代表一个顶点。根据你是在进行二维或三维绘图，你可以传递 2 或者 3。尽管我们的物体是存在于一个平面的，我们仍然将其绘制在三维虚拟世界中，因此每个顶点用三个数值表示，所以我们传递 3 给函数。然后，我们传递一个枚举值告诉 OpenGL 顶点是由 `GLfloat` 构成。OpenGL ES 允许你在本地数组中使用大部分的数据类型，但除 `GL_FLOAT` 外，其他都很少见。下一个参数... 现在不需要考虑下一个参数。那是以后讨论的主题。现在，它始终为 0。在以后的文章中，我将讨论怎样使用此参数将同一对象以不同的数据类型混杂在一个数据结构中。

随后，我们通知 OpenGL 通过刚才提交的顶点数组来绘制三角形。

```
glDrawArrays(GL_TRIANGLES, 0, 9);
```

你可能已经可以猜到，第一个枚举值是告诉 OpenGL 绘制什么。尽管 OpenGL ES 不支持绘制三角形之外的四边形或其他多边形，但它仍然支持一些其他绘图模式，如绘制点，线，线回路，三角形条和三角形扇。稍后我们将讨论这些绘图模式。

最后，我们要禁止先前启动了的特性以保证不会被其他地方的代码弄混。本例中没有其他的代码了，但通常你可以使用 OpenGL 绘制多个物体。

```
glDisableClientState(GL_VERTEX_ARRAY);
```

好了，我们的代码可以工作了尽管它不是那么引人入胜而且不是十分高效，但它确实确实可以工作。每秒钟我们的代码被调用数次。不相信？加入下列黑体代码再次运行：

```
– (void)drawView: (GLView*) view;
{
    static      GLfloat rotation = 0.0;
```



```

Vertex3D    vertex1 = Vertex3DMake(0.0, 1.0, -3.0);
Vertex3D    vertex2 = Vertex3DMake(1.0, 0.0, -3.0);
Vertex3D    vertex3 = Vertex3DMake(-1.0, 0.0, -3.0);
Triangle3D  triangle = Triangle3DMake(vertex1, vertex2, vertex3);

glLoadIdentity();
glRotatef(rotation, 0.0, 0.0, 1.0);
glClearColor(0.7, 0.7, 0.7, 1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnableClientState(GL_VERTEX_ARRAY);
glColor4f(1.0, 0.0, 0.0, 1.0);
glVertexPointer(3, GL_FLOAT, 0, &triangle);
glDrawArrays(GL_TRIANGLES, 0, 9);
glDisableClientState(GL_VERTEX_ARRAY);

    rotation+= 0.5;
}

```

当你再次运行时，三角形将沿着原点缓缓转动。先不需要关注太多旋转的逻辑。我只是想告诉你我们的代码每秒钟被调用了多次。

如果你想画正方形怎么办？OpenGL ES 并不支持正方形，所以我们只能通过三角形来定义正方形。这很简单 - 一个正方形可以通过两个三角形构成。我们要怎样调整上叙代码来绘制两个三角形？是不是可以创建两个 `Triangle3D`？是的，你可以这样做，但那没有效率。我们最好将两个三角形置入同一个顶点数组中。我们可以通过定义一个包含两个 `Triangle3D` 对象的数组，或分配大小等于两个 `Triangle3D` 对象或 18 个 `GLfloat` 的内存。

这是一种方法：

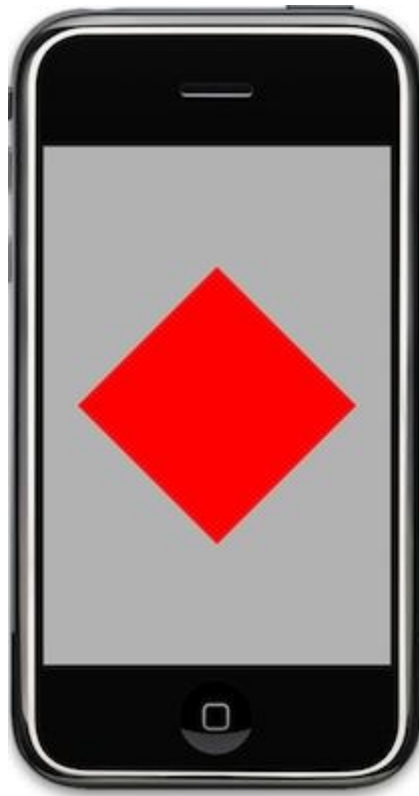
```

- (void)drawView: (GLView*)view;
{
    Triangle3D  triangle[2];
    triangle[0].v1 = Vertex3DMake(0.0, 1.0, -3.0);
    triangle[0].v2 = Vertex3DMake(1.0, 0.0, -3.0);
    triangle[0].v3 = Vertex3DMake(-1.0, 0.0, -3.0);
    triangle[1].v1 = Vertex3DMake(-1.0, 0.0, -3.0);
    triangle[1].v2 = Vertex3DMake(1.0, 0.0, -3.0);
    triangle[1].v3 = Vertex3DMake(0.0, -1.0, -3.0);

    glLoadIdentity();
    glClearColor(0.7, 0.7, 0.7, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnableClientState(GL_VERTEX_ARRAY);
    glColor4f(1.0, 0.0, 0.0, 1.0);
    glVertexPointer(3, GL_FLOAT, 0, &triangle);
    glDrawArrays(GL_TRIANGLES, 0, 18);
    glDisableClientState(GL_VERTEX_ARRAY);
}

```


运行，你将看到如下屏幕：



由于 `Vertex3DMake()` 方法在堆中创建新的 `Vertex3D` 然后复制其值到数组中而造成额外的内存被占用，因此上述代码并不理想。

对于这样简单的情况是没有什么问题的，但是在一个更为复杂的情况下，如果定义的 3D 物体很大时，那么你不会希望将其分配在堆中而且你不会希望对一个顶点不止一次地进行内存分配，所以最好是养成习惯通过我们的老朋友 `malloc()`（我更喜欢用 `calloc()`，因为它会将所有值设为 0，比较易于查找错误）将顶点分配在栈中。首先我们需要一个函数设定现存顶点的值而不是像 `Vertex3DMake()` 一样创建一个新对象。如下：

```
static inline void Vertex3DSet(Vertex3D *vertex, CGFloat inX, CGFloat inY, CGFloat inZ)
{
    vertex->x = inX;
    vertex->y = inY;
    vertex->z = inZ;
}
```

现在，我们使用新方法将两个三角形分配在栈中，重写代码：

```
- (void)drawView:(GLView*)view;
{
    Triangle3D *triangles = malloc(sizeof(Triangle3D) * 2);

    Vertex3DSet(&triangles[0].v1, 0.0, 1.0, -3.0);
    Vertex3DSet(&triangles[0].v2, 1.0, 0.0, -3.0);
    Vertex3DSet(&triangles[0].v3, -1.0, 0.0, -3.0);
    Vertex3DSet(&triangles[1].v1, -1.0, 0.0, -3.0);
    Vertex3DSet(&triangles[1].v2, 1.0, 0.0, -3.0);
    Vertex3DSet(&triangles[1].v3, 0.0, -1.0, -3.0);
}
```

```

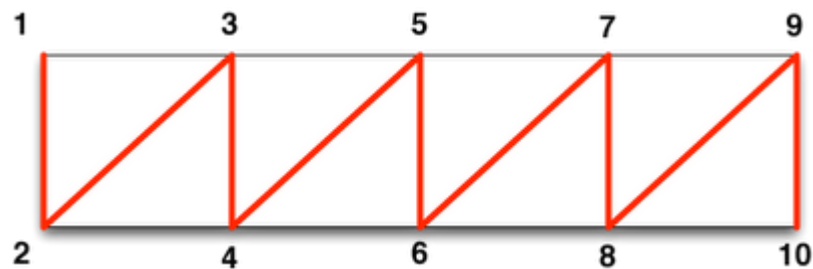
glLoadIdentity();
glClearColor(0.7, 0.7, 0.7, 1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnableClientState(GL_VERTEX_ARRAY);
glColor4f(1.0, 0.0, 0.0, 1.0);
glVertexPointer(3, GL_FLOAT, 0, triangles);
glDrawArrays(GL_TRIANGLES, 0, 18);
glDisableClientState(GL_VERTEX_ARRAY);

if (triangles != NULL)
    free(triangles);
}

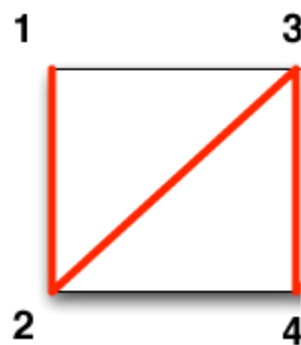
```

好了，我们已经讨论了许多基础知识，我们现在更深入一点。记住我说过 OpenGL ES 不止一种绘图方式吗？现在正方形需要 6 个顶点(18 个 GLfloat) 实际上我们可以使用 **triangle strips** (GL_TRIANGLE_STRIP)方法通过四个顶点(12 个 GLfloat)来绘制正方形。

这里是三角形条的基本概念：第一个三角形条是由前三个顶点构成(索引 0, 1, 2)。第二个三角形条是由前一个三角形的两个顶点加上数组中的下一个顶点构成，继续直到整个数组结束。看下图更清楚 - 第一个三角形由顶点 1, 2, 3 构成，下一个三角形由顶点 2, 3, 4 构成，等等：



所以，我们的正方形是这样构成的：



代码如下：

```

- (void)drawView: (GLView*)view;
{
    Vertex3D *vertices = malloc(sizeof(Vertex3D) * 4);

    Vertex3DSet(&vertices[0], 0.0, 1.0, -3.0);
    Vertex3DSet(&vertices[1], 1.0, 0.0, -3.0);

```

```

Vertex3DSet(&vertices[2], -1.0, 0.0, -3.0);
Vertex3DSet(&vertices[3], 0.0, -1.0, -3.0);

glLoadIdentity();

glClearColor(0.7, 0.7, 0.7, 1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnableClientState(GL_VERTEX_ARRAY);
glColor4f(1.0, 0.0, 0.0, 1.0);
glVertexPointer(3, GL_FLOAT, 0, vertices);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 12);
glDisableClientState(GL_VERTEX_ARRAY);

if (vertices != NULL)
    free(vertices);
}

```

我们再返回到第一段代码看看。记住我们是怎样绘制第一个三角形吗？我们使用 `glColor4f()` 设置颜色并且说设置的颜色一直适用于随后的代码。那意味着定义于顶点数组中的物体必须用同一种颜色绘制。很有局限性，对吗？

并非如此。正如 OpenGL ES 允许你将所有顶点置于一个数组中，它还允许你将每个顶点使用的颜色置于一个**颜色数组 (color array)** 中。如果你选择使用颜色数组，那么你需要为每个顶点设置颜色（四个 `GLfloat` 值）。通过下面方法启动颜色数组：

```
glEnableClientState(GL_COLOR_ARRAY);
```

我们可以象顶点数组一样定义一个包含四个 `GLfloat` 成员的 `Color3D` 结构。下面是怎样为原始三角形分配不同颜色的示例：

```

- (void)drawView:(GLView*)view;
{
    Vertex3D    vertex1 = Vertex3DMake(0.0, 1.0, -3.0);
    Vertex3D    vertex2 = Vertex3DMake(1.0, 0.0, -3.0);
    Vertex3D    vertex3 = Vertex3DMake(-1.0, 0.0, -3.0);
    Triangle3D  triangle = Triangle3DMake(vertex1, vertex2, vertex3);

    Color3D     *colors = malloc(sizeof(Color3D) * 3);
    Color3DSet(&colors[0], 1.0, 0.0, 0.0, 1.0);
    Color3DSet(&colors[1], 0.0, 1.0, 0.0, 1.0);
    Color3DSet(&colors[2], 0.0, 0.0, 1.0, 1.0);

    glLoadIdentity();
    glClearColor(0.7, 0.7, 0.7, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_COLOR_ARRAY);
    glColor4f(1.0, 0.0, 0.0, 1.0);
    glVertexPointer(3, GL_FLOAT, 0, &triangle);
}

```

```

glColorPointer(4, GL_FLOAT, 0, colors);
glDrawArrays(GL_TRIANGLES, 0, 9);
glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);

if (colors != NULL)
    free(colors);
}

```

运行，屏幕如下：



今天我们还要讨论一个话题。如果我们不止一次使用一个顶点（三角形条或三角形扇的相邻顶点除外），我们至今使用的方法存在一个问题，我们必须多次向 OpenGL 传递同一顶点。那不是什么大问题，但通常我们需要尽量减少向 OpenGL 传递的数据量，所以一次又一次地传递同一个 4 字节浮点数是非常地不理想。在一些物体中，某个顶点会用在七个甚至更多的不同三角形中，因此你的顶点数组可能会增大许多倍。

当处理这类复杂的几何体时，有一种方法可以避免多次传递同一个顶点，就是使用通过顶点对应于顶点数组中的索引的方法，此方法称之为**元素（elements）**。其原理是创建一个每个顶点只使用一次的数组。然后使用另一个使用最小的无符号整型数的数组来保存所需的唯一顶点号。换句话说，如果顶点数组具有小于 256 个顶点，那么你应该创建一个 GLubyte 数组，如果大于 256，但小于 65,536，应使用 GLushort。你可以通过映射顶点在第一个数组中的索引值来创建三角形（或其他形状）。所以，如果你创建了一个具有 12 个顶点的数组，那么数组中的第一个顶点为 0。你可以按以前一样的方法绘制图形，只不过不是调用 glDrawArrays()，而是调用不同的函数 glDrawElements() 并传递整数数组。

让我们以一个真实的，如假包换的 3D 形状来结束我们的教程：一个二十面体。每个人都使用正方体，但我们要更怪异一点，我们要画一个二十面体。替换 drawView：

```
- (void)drawView:(GLView*)view;
```

```

{

static GLfloat rot = 0.0;

// This is the same result as using Vertex3D, just faster to type and
// can be made const this way
static const Vertex3D vertices[] = {
    {0, -0.525731, 0.850651},          // vertices[0]
    {0.850651, 0, 0.525731},          // vertices[1]
    {0.850651, 0, -0.525731},         // vertices[2]
    {-0.850651, 0, -0.525731},        // vertices[3]
    {-0.850651, 0, 0.525731},         // vertices[4]
    {-0.525731, 0.850651, 0},         // vertices[5]
    {0.525731, 0.850651, 0},          // vertices[6]
    {0.525731, -0.850651, 0},         // vertices[7]
    {-0.525731, -0.850651, 0},        // vertices[8]
    {0, -0.525731, -0.850651},        // vertices[9]
    {0, 0.525731, -0.850651},         // vertices[10]
    {0, 0.525731, 0.850651}          // vertices[11]
};

static const Color3D colors[] = {
    {1.0, 0.0, 0.0, 1.0},
    {1.0, 0.5, 0.0, 1.0},
    {1.0, 1.0, 0.0, 1.0},
    {0.5, 1.0, 0.0, 1.0},
    {0.0, 1.0, 0.0, 1.0},
    {0.0, 1.0, 0.5, 1.0},
    {0.0, 1.0, 1.0, 1.0},
    {0.0, 0.5, 1.0, 1.0},
    {0.0, 0.0, 1.0, 1.0},
    {0.5, 0.0, 1.0, 1.0},
    {1.0, 0.0, 1.0, 1.0},
    {1.0, 0.0, 0.5, 1.0}
};

static const GLubyte icosahedronFaces[] = {
    1, 2, 6,
    1, 7, 2,
    3, 4, 5,
    4, 3, 8,
    6, 5, 11,
    5, 6, 10,

```

```

    9, 10, 2,
    10, 9, 3,
    7, 8, 9,
    8, 7, 0,
    11, 0, 1,
    0, 11, 4,
    6, 2, 10,
    1, 6, 11,
    3, 5, 10,
    5, 4, 11,
    2, 7, 9,
    7, 1, 0,
    3, 9, 8,
    4, 8, 0,
};

glLoadIdentity();
glTranslatef(0.0f, 0.0f, -3.0f);
glRotatef(rot, 1.0f, 1.0f, 1.0f);
glClearColor(0.7, 0.7, 0.7, 1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glVertexPointer(3, GL_FLOAT, 0, vertices);
glColorPointer(4, GL_FLOAT, 0, colors);

glDrawElements(GL_TRIANGLES, 60, GL_UNSIGNED_BYTE,
icosahedronFaces);

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
static NSTimeInterval lastDrawTime;
if (lastDrawTime)
{
    NSTimeInterval timeSinceLastDraw = [NSDate
timeIntervalSinceReferenceDate] - lastDrawTime;
    rot+=50 * timeSinceLastDraw;
}
lastDrawTime = [NSDate timeIntervalSinceReferenceDate];
}

```

运行，屏幕上将看到一个漂亮的旋转物体：



因为我们没有使用光源而且即使使用了光源我们也没有告诉 OpenGL 怎样进行光源反射,所以它看上去还不是一个完美的 3D 物体。有关光线的部分将在后续文章中讨论。

这里我们做了什么?首先,我们建立了一个静态变量来跟踪物体的旋转。

```
static GLfloat rot = 0.0;
```

然后我们定义顶点数组。我们使用了一个与前不同的方法,但结果是一样的。由于我们的几何体根本不会变化,所以我们将其定义为 `const`, 这样就不需要每一帧都分配/清除内存:

```
static const Vertex3D vertices[] = {  
    {0, -0.525731, 0.850651},           // vertices[0]  
    {0.850651, 0, 0.525731},           // vertices[1]  
    {0.850651, 0, -0.525731},          // vertices[2]  
    {-0.850651, 0, -0.525731},         // vertices[3]  
    {-0.850651, 0, 0.525731},          // vertices[4]  
    {-0.525731, 0.850651, 0},          // vertices[5]  
    {0.525731, 0.850651, 0},           // vertices[6]  
    {0.525731, -0.850651, 0},          // vertices[7]  
    {-0.525731, -0.850651, 0},         // vertices[8]  
    {0, -0.525731, -0.850651},         // vertices[9]  
    {0, 0.525731, -0.850651},          // vertices[10]  
    {0, 0.525731, 0.850651},           // vertices[11]  
};
```

然后用同样方法建立一个颜色数组。创建一个 `Color3D` 对象数组,每项对应于前一个数组的顶点:

```
static const Color3D colors[] = {  
    {1.0, 0.0, 0.0, 1.0},
```



```

        {1.0, 0.5, 0.0, 1.0},
        {1.0, 1.0, 0.0, 1.0},
        {0.5, 1.0, 0.0, 1.0},
        {0.0, 1.0, 0.0, 1.0},
        {0.0, 1.0, 0.5, 1.0},
        {0.0, 1.0, 1.0, 1.0},
        {0.0, 0.5, 1.0, 1.0},
        {0.0, 0.0, 1.0, 1.0},
        {0.5, 0.0, 1.0, 1.0},
        {1.0, 0.0, 1.0, 1.0},
        {1.0, 0.0, 0.5, 1.0}
    };

```

最后，创建二十面体。上述十二个顶点本身并未描述形状。OpenGL 需要知道怎样将它们联系在一起，所以我们创建了一个整型数组 (GLubyte) 指向构成各三角形的顶点。

```

static const GLubyte icosahedronFaces[] = {
    1, 2, 6,
    1, 7, 2,
    3, 4, 5,
    4, 3, 8,
    6, 5, 11,
    5, 6, 10,
    9, 10, 2,
    10, 9, 3,
    7, 8, 9,
    8, 7, 0,
    11, 0, 1,
    0, 11, 4,
    6, 2, 10,
    1, 6, 11,
    3, 5, 10,
    5, 4, 11,
    2, 7, 9,
    7, 1, 0,
    3, 9, 8,
    4, 8, 0,
};

```

二十面体的第一个面的三个数是 1,2,6，代表绘制处于索引 1 (0.850651, 0, 0.525731)， 2 (0.850651, 0, 0.525731)， 和 6 (0.525731, 0.850651, 0)之间的三角形。

下面一段代码没有新内容，只是加载单元矩阵（所以变换复位），移动并旋转几何体，设置背景色，清除缓存，启动顶点和颜色数组，然后提供顶点数组数据给 OpenGL。所有这些在前一个例子中都有涉及。

```

glLoadIdentity();
glTranslatef(0.0f, 0.0f, -3.0f);
glRotatef(rot, 1.0f, 1.0f, 1.0f);

```

```

glClearColor(0.7, 0.7, 0.7, 1.0);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);
glColor4f(1.0, 0.0, 0.0, 1.0);
glVertexPointer(3, GL_FLOAT, 0, vertices);
glColorPointer(4, GL_FLOAT, 0, colors);

```

然后，我们没有使用 `glDrawArrays()`。而是调用 `glDrawElements()`：

```

glDrawElements(GL_TRIANGLES, 60, GL_UNSIGNED_BYTE,
icosahedronFaces);

```

接着，执行禁止功能，根据距上一帧绘制的时间增加旋转变量值：

```

glDisableClientState(GL_VERTEX_ARRAY);
glDisableClientState(GL_COLOR_ARRAY);
static NSTimeInterval lastDrawTime;
if (lastDrawTime)
{
    NSTimeInterval timeSinceLastDraw = [NSDate
timeIntervalSinceReferenceDate] - lastDrawTime;
    rot+=50 * timeSinceLastDraw;
}
lastDrawTime = [NSDate timeIntervalSinceReferenceDate];

```

记住：如果你按绘制的正确次序提供顶点，那么你应该使用 `glDrawArrays()`，但是如果你提供一个数组然后用另一个以索引值区分顶点次序的数组的话，那么你应该使用 `glDrawElements()`。

请花些时间测试绘图代码，添加更多的多边形，改变颜色等。OpenGL 绘图功能远超过本文涉及的内容，但你应该清楚 iPhone 上 3D 物体绘制的基本概念了：创建一块内存保存所有的顶点，传递顶点数组给 OpenGL，然后由 OpenGL 绘制出来。

三. 透视

现在你已经知道 OpenGL 是怎样绘图的了，让我们回头谈谈一个很重要的概念：OpenGL 视口 (**viewport**)。许多人对 3D 编程还很陌生，那些使用过像 Maya, Blender, 或 Lightwave 之类 3D 图形程序的人都试图在 OpenGL 虚拟世界中找到“摄像机”。但 OpenGL 并不存在这样的东西。它所有的是在 3D 空间中定义可见的物体。虚拟世界是没有边界的，但计算机不可能处理无限的空间，所以 OpenGL 需要我们定义一个可以被观察者看到的空间。

如果我们从大部分 3D 程序具有的摄像机对象的角度出发来考虑，视口端点的中心就是摄像机。也就是观察者站的位置。它是一个观察虚拟世界的虚拟窗口。观察者可见的空间有一定限制。她看不见她身后的东西。她也看不见视角之外的东西。而且她还不能看见太远的东西。可以认为视口是通过“观察者可见”参数所确定的形状。很简单，对吗？

不幸的是，并非如此。要解释原因，我们首先需要讨论的是在 OpenGL ES 中具有两种不同的视口类型：正交和透视。

正交和透视

为更好地理解，我们先看看铁路轨道，好吗？要正常工作，铁路的两条铁轨之间必须具有固定的距离。其固定的距离是由铁轨根据承载什么样的火车而决定。重要的是铁轨（以及火车的轮子）必须具有相同的距离。如果不是这样，火车根本不可能运行。

如果我们从上方观察铁轨，这个事实很明显。



但是如果你站在铁轨上向下观察会怎么样。不要说“你会被火车撞”，我假设你会足够聪明，会在没有火车开动时进行观察。



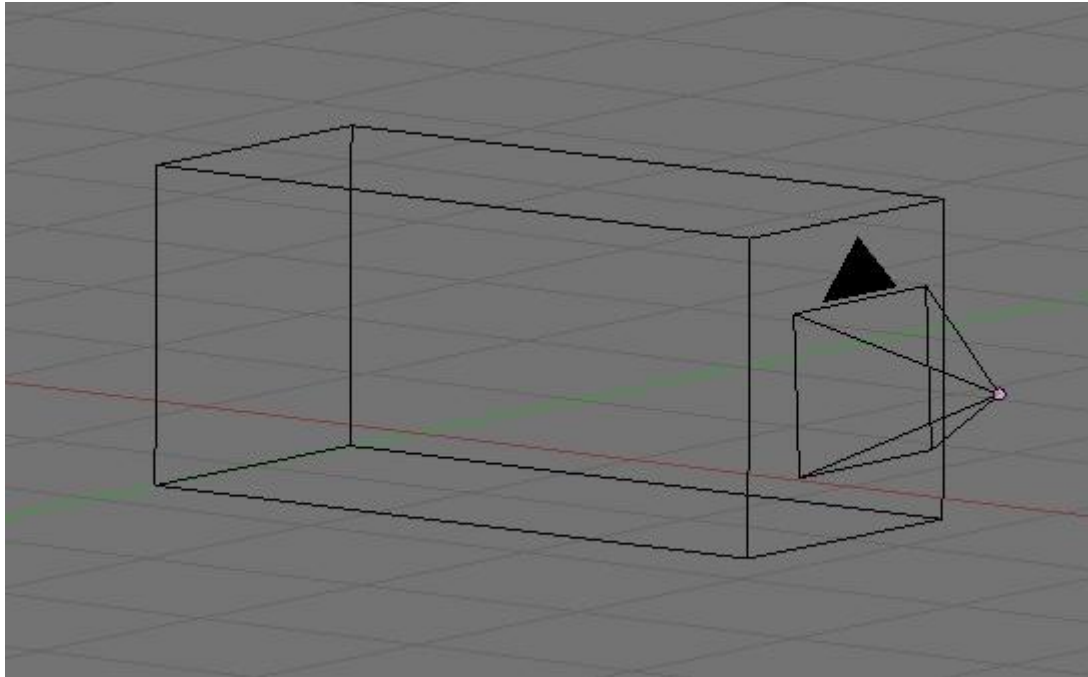
是的，铁轨看上去越远与靠近。感谢二年级美术老师，可能你已经知道这就是所谓**透视 (perspective)**。

OpenGL 可以设定的视口中的一种就是使用透视。当你这样设置视口时，物体会随着移远而越来越小，视线会在物体远离观察者时最终交汇。这是对真实视觉的模拟；人们就是以这种方式观察世界的。

另一种看设置的视口称为**正交 (orthogonal)** 视口。这种类型的视口，视线永远不会交汇而且物体不会改变其大小。没有透视效果。对于 CAD 程序以及其他各种目的是十分方便的，但因为它不像人们眼睛观察的方式所以看上去是不真实的，通常也不是你所希望的。

对于正交视口，你看将摄像机置于铁轨上，但这些铁轨永远不会交汇。它们将随着远离你的视线而继续保持等距。即使你定义了一个无限大的视口（OpenGL ES 并不支持），这些线仍保持等距。

正交视口的优点是容易定义。因为线永不交汇，你只需定义一个像箱子一样的 3D 空间，像这样：



设置正交视口

在使用 `glViewport()` 函数定义视口前，你可以通过 `glOrthof()` 通知 OpenGL ES 你希望使用正交视口。下面是一个简单的例子：

```
CGRect rect = view.bounds;
glOrthof(-1.0,                                // Left
          1.0,                                // Right
          -1.0 / (rect.size.width / rect.size.height), // Bottom
          1.0 / (rect.size.width / rect.size.height), // Top
          0.01,                                // Near
          10000.0);                             // Far

glViewport(0, 0, rect.size.width, rect.size.height);
```

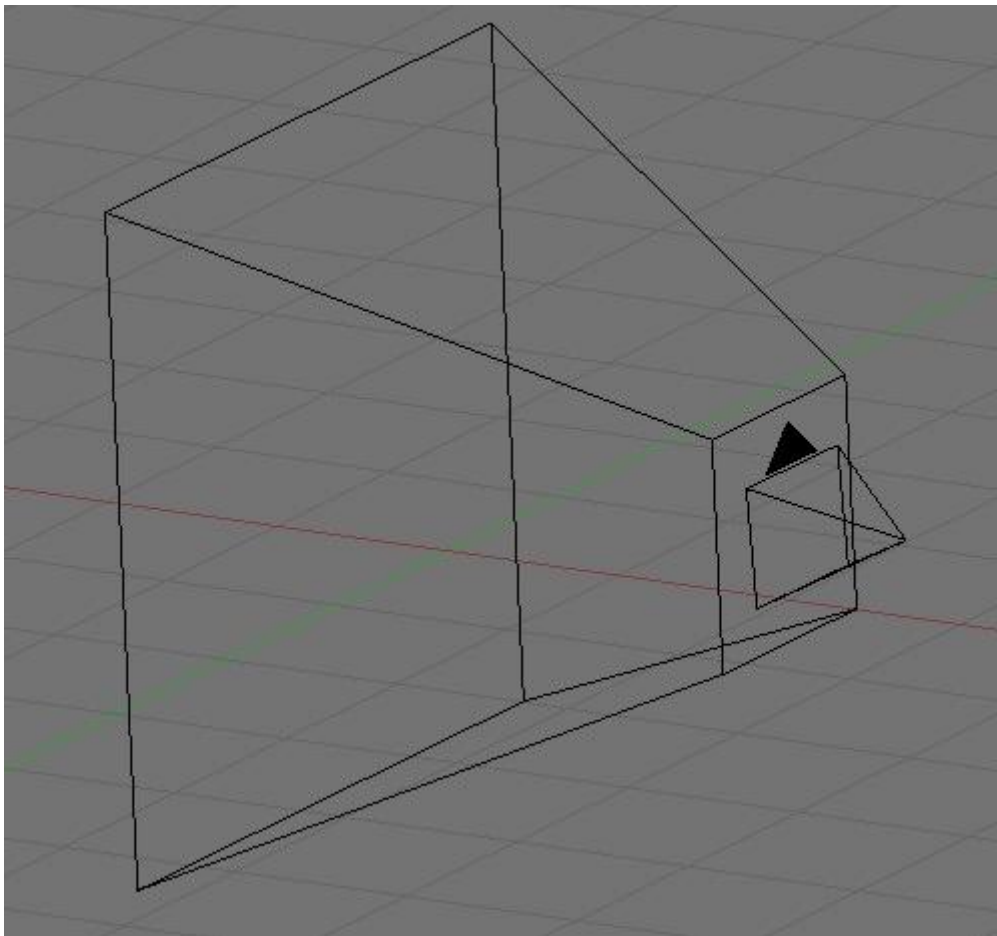
这不难理解。首先我们获取视窗的尺寸。然后设定视口空间的宽度为两个单位，沿 x 轴从 -1.0 到 +1.0。很容易吧。接着怎样设定底部和顶部？我们希望我们定义空间的 X 和 Y 坐标的宽高比与视窗的宽高比（也就是 iPhone 全屏时的宽高比）一样。由于 iPhone 的宽度与高度不同，我们需要确保视口的 x 和 y 坐标不同，但遵循一样的比例。

之后，我们定义了 `near`（远）和 `far`（近）范围来描述观察的深度。`near` 参数说明了视口开始的位置。如果我们站在原点处，视口就位于我们的面前，所以习惯上使用 `.01` 或 `.001` 作为正交视口的起点。这使得视口处于原点“前方”一点点。`far` 可以根据你程序的需要来设定。如果你程序中的物体永远不会远过 20 个单位，那么你不需要将 `far` 设置为 20,000 个单位。具体的数字随程序的不同而不同。

调用 `glOrthof()` 之后，我们使用视窗矩形来调用 `glViewport()`。
这是比较简单的情況。

设置透视视口

另一种情况就不那么简单，这里是原因。如果物体随着远离观察者而变小，那么它和你定义的可见空间的形状有什么关系。随着视线越来越远，你可以看到更广阔的世界，所以如果你使用透视，那么你定义的空间将不是一个立方体。是的，当使用透视时可见空间的形状称为**锥台 (frustum)**。是的，我知道，奇怪的名字。但却是真实的。我们的锥台看上去像这样：



请注意当我们离视口越来越远时（换句话说，当 z 值减小时），观察体的 x 和 y 坐标都会越来越大。

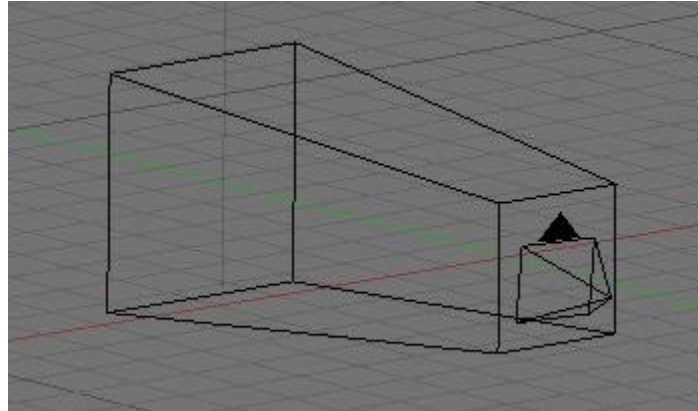
要设置透视视口，我们不使用 `glOrthof()`，我们使用一个不同的函数 `glFrustumf()`。此函数使用同样的六个参数。很容易理解，但我们应该怎样确定传递给 `glFrustumf()` 的参数？

`near` 和 `far` 容易理解。你可以同样方式理解它们。`near` 使用类似 `.001` 的数值，然后根据不同程序的需要确定 `far` 值。

但是 `left`, `right`, `bottom`, 和 `top` 呢？为设置这些值，我们需要一点点数学计算。

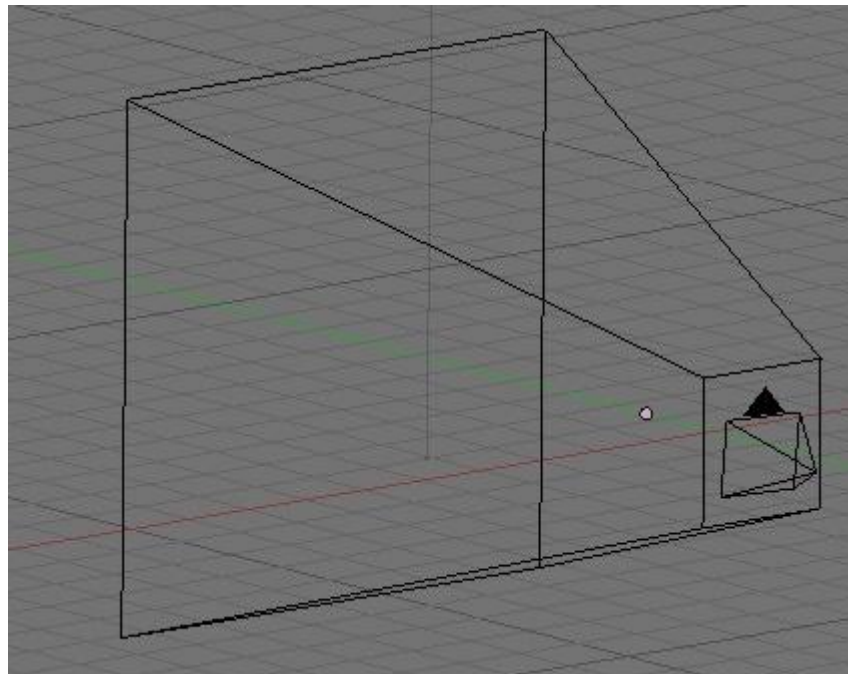
要计算锥台，我们首先要理解**视野 (field of vision)** 的概念，它是由两个角度定义的。让我们这样做：伸出双臂手掌合拢伸向前方。你的手臂现在指向你自己锥台的 z 轴，对吗？好，现在慢慢分开你的双臂。由于在你双臂展开时肩膀保

持不动，你定义了一个逐渐增大的角度。这就是用于定义观察锥台的两个角度之一。它定义了视野的宽度。另一个角度的定义原理一样，只是这次你向上下展开你的双臂。如果你的双手间距只有三英寸，那么 x 角度将非常小。



这称为窄视野。

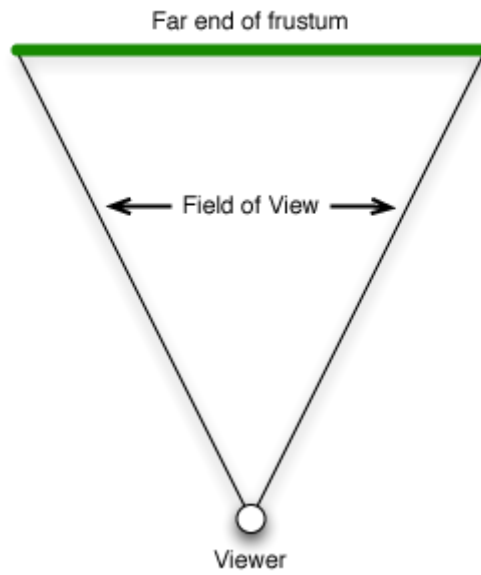
如果你双手分开两英尺，视野的宽度变得很大。



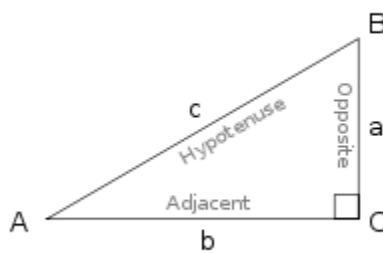
这就是所谓 宽视角（广角）。

如果用摄影术语描述，你可将视野当作虚拟相机的虚拟光圈的焦距。窄视野很像摄远镜头，它造就了一个缓慢增长的长锥台。宽视角就像广角镜，它造就了一个增长很快的锥台。

我们选择一个中间值，例如 45° 。使用这个值，我们怎样计算我们的观察锥台？我们先看下两个角度中的一个。想象一下，从顶部看锥台是什么样子。下面是示意图：

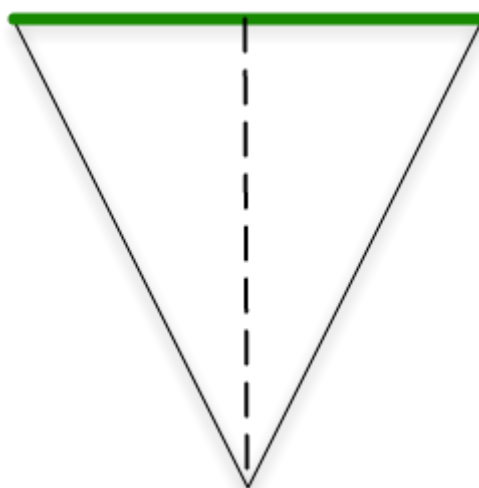


从上向下看，它就像一个砍掉一个点的三角形。但对我们而言，它已经足够接近一个三角形。你还记得三角课上的正切吗？正切函数定义为直角对边与相邻边的比率。



但是，我们没有直角，是吗？

实际上，我们有两个直角... 如果我们沿 z 轴向下画一条直线的话：



中心虚线就是两个直角的“相邻边”。所以，锥台远端宽度的一半就是视野角度正切的一半。如果我们将此值乘以 `near` 值，就可以得到 `right` 值。`right` 值取反就是 `left`。

我们希望视野具有与屏幕一样的长宽比，所以按照 `glOrthof()` 中相同的方法（将 `right` 乘以屏幕的长宽比）来计算 `top` 和 `bottom` 值。代码如下：

```
CGRect rect = view.bounds;
```

```
GLfloat size = .01 * tanf(DEGREES_TO_RADIANS(45.0) / 2.0);
```

```
glFrustumf(-size,                                // Left
            size,                                  // Right
            -size / (rect.size.width / rect.size.height), // Bottom
            size / (rect.size.width / rect.size.height),  // Top
            .01,                                          // Near
            1000.0);                                     // Far
```

注意：关于 `glFrustum()` 怎样使用传递的参数计算锥台的形状将在我们讨论矩阵时讨论。现在，我们暂且相信计算是正确的，好吗？

让我们运用到程序中。我修改了上篇文章中最终的 `drawView:` 方法，我们将沿 `z` 轴向下显示了三十个二十面体。下面是新的 `drawView:` 方法：

```
- (void)drawView:(GLView*)view;
```

```
{
```

```
    static GLfloat rot = 0.0;
```

```
    static const Vertex3D vertices[] = {
```

```
        {0, -0.525731, 0.850651},           // vertices[0]
```

```
        {0.850651, 0, 0.525731},           // vertices[1]
```

```
        {0.850651, 0, -0.525731},          // vertices[2]
```

```
        {-0.850651, 0, -0.525731},         // vertices[3]
```

```
        {-0.850651, 0, 0.525731},          // vertices[4]
```

```
        {-0.525731, 0.850651, 0},          // vertices[5]
        {0.525731, 0.850651, 0},          // vertices[6]
        {0.525731, -0.850651, 0},         // vertices[7]
        {-0.525731, -0.850651, 0},        // vertices[8]
        {0, -0.525731, -0.850651},        // vertices[9]
        {0, 0.525731, -0.850651},         // vertices[10]
        {0, 0.525731, 0.850651}           // vertices[11]
    };
```

```
static const Color3D colors[] = {
```

```
    {1.0, 0.0, 0.0, 1.0},
    {1.0, 0.5, 0.0, 1.0},
    {1.0, 1.0, 0.0, 1.0},
    {0.5, 1.0, 0.0, 1.0},
    {0.0, 1.0, 0.0, 1.0},
    {0.0, 1.0, 0.5, 1.0},
    {0.0, 1.0, 1.0, 1.0},
    {0.0, 0.5, 1.0, 1.0},
    {0.0, 0.0, 1.0, 1.0},
    {0.5, 0.0, 1.0, 1.0},
    {1.0, 0.0, 1.0, 1.0},
    {1.0, 0.0, 0.5, 1.0}
```

```
};
```

```
static const GLubyte icosahedronFaces[] = {
```

```
    1, 2, 6,
```

```
    1, 7, 2,
```

```
    3, 4, 5,
```

```
    4, 3, 8,
```

```
    6, 5, 11,
```

```
    5, 6, 10,
```

```
    9, 10, 2,
```

```
    10, 9, 3,
```

```
    7, 8, 9,
```

```
    8, 7, 0,
```

```
    11, 0, 1,
```

```
    0, 11, 4,
```

```
    6, 2, 10,
```

```
    1, 6, 11,
```

```
    3, 5, 10,
```

```
    5, 4, 11,
```

```
    2, 7, 9,
```

```
    7, 1, 0,
```

```
    3, 9, 8,
```

```

        4, 8, 0,

};

glLoadIdentity();

glClearColor(0.7, 0.7, 0.7, 1.0);

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glEnableClientState(GL_VERTEX_ARRAY);

glEnableClientState(GL_COLOR_ARRAY);

glVertexPointer(3, GL_FLOAT, 0, vertices);

glColorPointer(4, GL_FLOAT, 0, colors);

for (int i = 1; i <= 30; i++)
{
    glLoadIdentity();

    glTranslatef(0.0f, -1.5, -3.0f * (GLfloat)i);

    glRotatef(rot, 1.0, 1.0, 1.0);

    glDrawElements(GL_TRIANGLES, 60, GL_UNSIGNED_BYTE,
icosahedronFaces);

}

glDisableClientState(GL_VERTEX_ARRAY);

glDisableClientState(GL_COLOR_ARRAY);

static NSTimeInterval lastDrawTime;

if (lastDrawTime)

```

```

{
    NSTimeInterval timeSinceLastDraw = [NSDate
timeIntervalSinceReferenceDate] - lastDrawTime;

    rot += 50 * timeSinceLastDraw;

}

lastDrawTime = [NSDate timeIntervalSinceReferenceDate];
}

```

如果你把上述代码加入 [OpenGL Xcode 项目模板](#) 项目中(它使用 `glFrustumf()` 设置了一个具有 45° 视野的透视视口), 你将看到下面图形 :



很好。随着几何体远离你, 它们会变得越来越小, 正像火车铁轨一样。

如果你只是将 `glFrustumf()` 改为 `glOrthof()`, 看上去就完全不同了 :



没有透视，第一个二十面体后面的二十九个二十面体完全被第一个挡住了。因为没有透视，后面的各几何体的形状完全取决于其前方的物体。

好了，这是一个很沉闷的主题，事实上你现在可以完全忘却三角课学的知识了。只要复制基于视野角度计算锥台的两行代码就好，而且你可能再也不需要记住它的原理了。

下一篇文章，我们将为二十面体增加光效，使它看上去更真实。



四. 光效

继续我们的 iPhone OpenGL ES 之旅，我们将讨论光效。目前，我们没有加入任何光效。幸运的是，OpenGL 在没有设置光效的情况下仍然可以看见东西。它只是提供一种十分单调的整体光让我们看到物体。但是如果不定义光效，物体看上去都很单调，就像你在[第二部分程序](#)中看到的那样。



阴影模型（Shade Model）

在深入讨论 OpenGL ES 是怎样处理光线之前，重要的是要了解 OpenGL ES 实际上定义了两种 **shade model**，`GL_FLAT` 和 `GL_SMOOTH`。我们将不会讨论 `GL_FLAT`，因为这只会让你的程序看上去来自九十年代：



GL_FLAT 方式渲染的一个二十面体。15 年前的实时渲染技术

从发光的角度来看，GL_FLAT 将指定三角形上的每个像素都同等对待。多边形上的每个像素都具有相同的颜色，阴影等。它提供了足够的视觉暗示使其看上去有立体感而且它的计算比每个像素按不同方法计算更为廉价，但是在这种方式下，物体看上去极为不真实。现在有人使用它可能是为了产生特殊的复古效果，但要使你的 3D 物体尽量真实，你应该使用 GL_SMOOTH 绘图模式，它使用了一种平滑但较快速的阴影算法，称为 [Gouraud](#) 算法。GL_SMOOTH 是默认值。

启动光效

我假定你继续使用第二部分的最终项目，即那个看上去不是很立体的旋转二十面体的项目。如果你手头上还没有那个项目，在[这里](#)下载。

第一件事就是要启动光效。默认情况下，手工指定光效是被禁止的。现在我们打开这项功能。在 GLViewController.m 的 setupView: 方法中加入黑体部分：

```
-(void)setupView: (GLView*)view
{
    const GLfloat zNear = 0.01, zFar = 1000.0, fieldOfView = 45.0;
    GLfloat size;
```

```

    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    size = zNear * tanf(DEGREES_TO_RADIANS(fieldOfView) / 2.0);
    CGRect rect = view.bounds;
    glFrustumf(-size, size, -size / (rect.size.width / rect.size.height),
size /
                (rect.size.width / rect.size.height), zNear, zFar);
    glViewport(0, 0, rect.size.width, rect.size.height);
    glMatrixMode(GL_MODELVIEW);

    glEnable(GL_LIGHTING);

    glLoadIdentity();
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
}

```

通常情况下，光效只需在设定时启动一次。不需要在绘图开始前后打开和关闭。可能有些特效的情况需要在程序执行时打开或关闭，但是大部分情况下，你只需在程序启动时打开它。此单行代码就是在 OpenGL ES 中启动光效。运行时会怎样？



启动光效

我们启动了光效，但是没有创建任何光源。除清除缓存用的灰色外任何绘制的物体都被渲染成绝对的黑色。没有太多的改进，对吗？让我们在场景中加入光源。

启动光效的方式有些奇怪。OpenGL ES 允许你创建 8 个光源。有一个常量对应于这些光源中的一个，常量为 `GL_LIGHT0` 到 `GL_LIGHT7`。可以任意组合这些光源中的五个，尽管习惯上从 `GL_LIGHT0` 作为第一个光源，然后是 `GL_LIGHT1` 等等。下面是“打开”第一个光源 `GL_LIGHT0` 的方法：

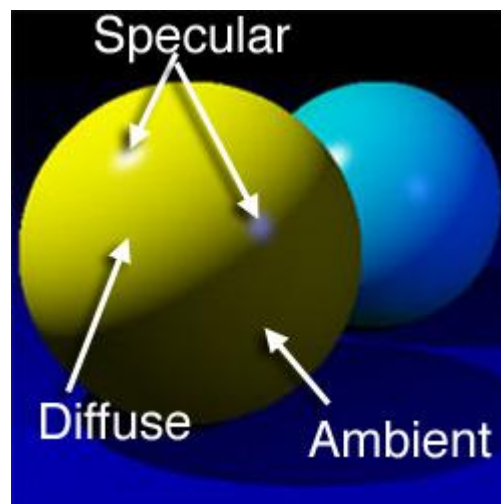
```
glEnable(GL_LIGHT0);
```

一旦你启动了光源，你必须设置光源的一些属性。作为初学者，有三个不同的要素用来定义光源。

光效三要素

在 OpenGL ES 中，光由三个元素组成，分别是**环境元素 (ambient component)**，**散射元素 (diffuse component)** 和 **高光元素 (specular component)**。我们使用颜色来设定光线元素，这看上去有些奇怪，但是由于它允许你同时指定各光线元素的颜色和相对强度，这个方法工作得很好。明亮的白色光定义为白色 (`{1.0, 1.0, 1.0, 1.0}`)，而暗白色可能定义为灰色 (`{0.3, 0.3, 0.3, 1.0}`)。你还可以通过改变红，绿，蓝元素的百分比来调整色偏。

下图说明了各要素产生的效果。



高光元素定义了光线直接照射并反射到观察者从而形成了物体上的“热点”或光泽。光点的大小取决于一些因素，但是如果你看到如上图黄球所示一个区域明显的光斑，那通常就是来自于一个或多个光源的高光部分。

散射元素定义了比较平均的定向光源，在物体面向光线的一面具有光泽。

环境光则没有明显的光源。其光线折射与许多物体，因此无法确定其来源。环境元素平均作用于场景中的所有物体的所有面。

环境光

你的光效中有越多的环境元素，那么就越不会产生引入注目的效果。所有光线的环境元素会融合在一起产生效果，意思是场景中的总环境光效是由所有启动光源的环境光组合在一起所决定的。如果你使用了不止一个光源，那么最好是只指定一个光源的环境元素，而设定其他所有光源的环境因素为黑 $(\{0.0, 0.0, 0.0, 1.0\})$ ，从而很容易地调整场景的环境光效。

下面演示了怎样指定一个很暗的白色光源：

```
const GLfloat light0Ambient[] = {0.05, 0.05, 0.05, 1.0};  
glLightfv(GL_LIGHT0, GL_AMBIENT, light0Ambient);
```

使用像这样的很低的环境元素值使场景看上去更引人注目，但同时也意味着物体没有面向光线的面或者有其他物体挡住的物体将在场景中看得不是很清楚。

散射光

在 OpenGL ES 中可以设定的第二个光线元素是 **散射元素 (diffuse component)**。在现实世界里，散射光线是诸如穿透光纤或从一堵白墙反射的光线。散射光线是发散的，因而参数较柔和的光，一般不会像直射光一样产生光斑。如果你曾经观察过职业摄影家使用摄影室灯光，你可能会看到他们使用[柔光箱](#)或者反光伞。两者都会穿透像白布之类的轻型材料并反射与轻型有色材料从而使光线发散以产生令人愉悦的照片。在 OpenGL ES 中，散射元素作用类似，它使光线均匀地散布到物体之上。然而，不像环境光，由于它是定向光，只有面向光线的物体面才会反射散射光，而场景中的所有多面体都会被环境光照射。

下面的例子演示了设定场景中的第一个散射元素：

```
const GLfloat light0Diffuse[] = {0.5, 0.5, 0.5, 1.0};  
glLightfv(GL_LIGHT0, GL_DIFFUSE, light0Diffuse);
```

高光

最后，我们讨论高光。这种类型的光是十分直接的，它们会以热点和光晕的形式反射到观察者的眼中。如果你想产生聚光灯的效果，那么应该设置一个很大的高光元素值及很小的散射和环境元素值（还需要定义其他一些参数，等下会有介绍）。

注意: 在下一篇文章中你将看到，光线的高光值是确定高光尺寸的唯一因素。

下面是设定高光元素的例子：

```
const GLfloat light0Specular[] = {0.7, 0.7, 0.7, 1.0};
```

位置

还需要设定光效的另一个重要属性，即光源 3D 空间中的位置。这不会影响环境元素，但其他两个元素由于其本性，只有在 OpenGL 知道了场景中物体与光的相对位置后才能计算。例如：

```
const GLfloat light0Position[] = {10.0, 10.0, 10.0, 0.0};  
glLightfv(GL_LIGHT0, GL_POSITION, light0Position);
```

此位置将第一个光源放置在观察者后方的右上角。

这些是用于设定几乎所有光线的属性。如果你没有设定其中一个元素，那么它就采用默认值黑色 {0.0, 0.0, 0.0, 1.0}。如果你没有定义位置，那么它就处于原点，通常这不是你想要的结果。

你可能想知道 alpha 值对光线的作用。对环境光和高光而言这是个愚蠢的问题。然而在计算散射光确定光线是怎样反射时，需要用到它。我们将在讨论材质时再解释它是怎样工作的，因为材质和光线值都将出现在方程式中。我们下次再讨论材质，现在将 alpha 设为 1.0。改变其值对本文的程序不会产生任何影响，但有可能对以后的程序至少是有关散射元素的部分产生影响。

还有一些光线元素你可选择使用。

创建点光源（聚光灯）

如果你希望创建一个定向点光源（一种指向特定方向并照亮特定角度范围的光源，本质上，它与灯泡照亮各个方向相反，它只照亮一个锥台的范围），那么你需要设定两个额外参数。设定 GL_SPOT_DIRECTION 允许你指定光照的方向，它类似于上一篇文章中介绍的视野角度的计算。窄角度将产生很小范围的点光源，而宽角度则产生像泛光灯一样的效果。

指定光的方向

通过指定定义了光线指向的 x, y 和 z 值来使 GL_SPOT_DIRECTION 的工作。然而，光线并不指向你在空间中定义的那一点。你提供的三个坐标值是向量（vector），而非顶点。这是一个很细微但十分重要的区别。一个代表向量的数据结

构与一个顶点的数据结构完全一样（都有三个 GLfloats，其中每一个分别是笛卡尔的一个轴）。然而，向量的数据是用来表示方向而不是空间中的一点。

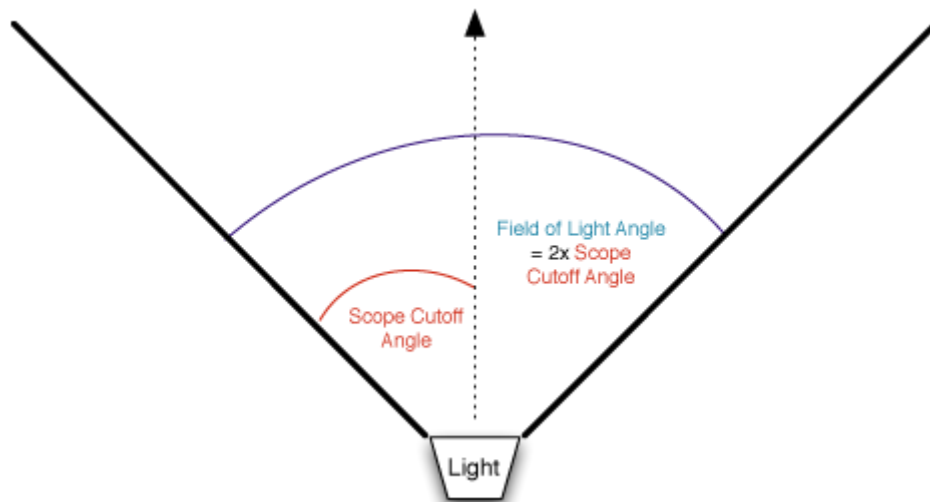
每个人都知道两点可以定义一条线段，那么空间中的一点怎么可能指定方向？这是因为存在一个隐性的第二点作为起点，即原点。如果你从原点画一条线到向量定义的点，那就是向量代表的方向。向量还可用于表示速度和距离，一个远离原点的点表示速度越快或距离更远。在大部分的 OpenGL 应用中，并未使用距离原点的距离。实际上，在大部分使用向量的情况下，我们需要将向量标准化（normalize）为长度 1.0。关于向量的标准化，随着我们继续深入将会讨论到。现在你只需知道，如果你希望定义一个定向光，那么你必须创建一个定义了光的方向的向量。下例演示了定义一个沿 z 轴而下的光源：

```
const GLfloat light0Direction = {0.0, 0.0, -1.0};  
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, light0Direction);
```

现在，如果你希望光线指向一个特定物体，应该怎么办？实际上很简单，将光的位置和物体的位置传入 OpenGLCommon.h 的函数 Vector3DMakeWithStartAndEndPoints() 中，它将返回一个被光照射到的指定点的标准化向量。然后，再将其作为 GL_SPOT_DIRECTION 的值。

指定光的角度

除非你限制光照的角度，否则指定光的方向并不会产生显著的效果。因为当你指定 GL_SPOT_CUTOFF 值时，它定义了中心线两边的角度，所以如果你指定截止角时，它必须小于 180°。如果你的定义为 45°，那么实际上你创建了一个总角度为 90°的点光源。这意味着你可设定的 GL_SPOT_CUTOFF 的最大值为 180°。下图说明了这个概念：



下例演示了怎样限制角度为 90°(使用 45° 截止角):

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
```

还有三个光的属性可以设置。它们是配合使用的，这些超出了本文的范围。以后，我可能在有关**光衰减**（光线随着远离光源而减弱）的文章中讨论到。通过调整衰减值可以产生很漂亮的效果。

综合

让我们综合所学内容在 `setupView:` 方法中设定一个光源。使用下列代码替换 `setupView:` 方法中原有的代码：

```
-(void) setupView: (GLView*) view
{
    const GLfloat zNear = 0.01, zFar = 1000.0, fieldOfView = 45.0;
    GLfloat size;
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    size = zNear * tanf(DEGREES_TO_RADIANS(fieldOfView) / 2.0);
    CGRect rect = view.bounds;
    glFrustumf(-size, size, -size / (rect.size.width / rect.size.height),
size /
                (rect.size.width / rect.size.height), zNear, zFar);
    glViewport(0, 0, rect.size.width, rect.size.height);
    glMatrixMode(GL_MODELVIEW);

    // Enable lighting
    glEnable(GL_LIGHTING);

    // Turn the first light on
    glEnable(GL_LIGHT0);

    // Define the ambient component of the first light
    const GLfloat light0Ambient[] = {0.1, 0.1, 0.1, 1.0};
    glLightfv(GL_LIGHT0, GL_AMBIENT, light0Ambient);

    // Define the diffuse component of the first light
    const GLfloat light0Diffuse[] = {0.7, 0.7, 0.7, 1.0};
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light0Diffuse);

    // Define the specular component and shininess of the first light
    const GLfloat light0Specular[] = {0.7, 0.7, 0.7, 1.0};
    const GLfloat light0Shininess = 0.4;
    glLightfv(GL_LIGHT0, GL_SPECULAR, light0Specular);

    // Define the position of the first light
    const GLfloat light0Position[] = {0.0, 10.0, 10.0, 0.0};
    glLightfv(GL_LIGHT0, GL_POSITION, light0Position);
```



```

    // Define a direction vector for the light, this one points right down
the Z axis
    const GLfloat light0Direction[] = {0.0, 0.0, -1.0};
    glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, light0Direction);

    // Define a cutoff angle. This defines a 90° field of vision, since
the cutoff
    // is number of degrees to each side of an imaginary line drawn from
the light's
    // position along the vector supplied in GL_SPOT_DIRECTION above
    glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);

    glLoadIdentity();
    glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
}

```

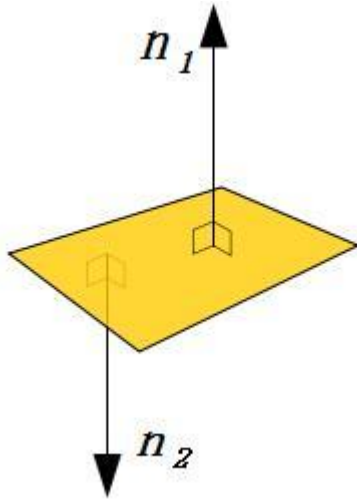
很简单吧？应该一切都准备好了吧？好，我们试着运行一下看看。



这是什么？太糟糕了吧！我们设定了光源，可是看不到任何东西啊？屏幕上只是显示一个黑和灰的形状，它甚至看上去不像个 3D 形状，比以前还糟糕。

不要紧张，这很正常（注： `It's Normal` 在这里有两层意思，一是正常，二是法线）

Normal 数学上的意思是“垂直于”。这是我们目前缺少的。法线。一个背面的法线（或多边形的法线）是一个垂直于指定多边形表面的向量（或直线）。参考下图：



OpenGL 渲染一个形状时并不需要知道法线，但在你使用定向光线时需要用到。OpenGL 需要表面法线来确定光线是怎样与各多边形交互作用的。

OpenGL 要求我们为各使用的顶点提供法线。计算一个三角形的表面法线是很简单的，它是三角形两边的叉积。代码如下：

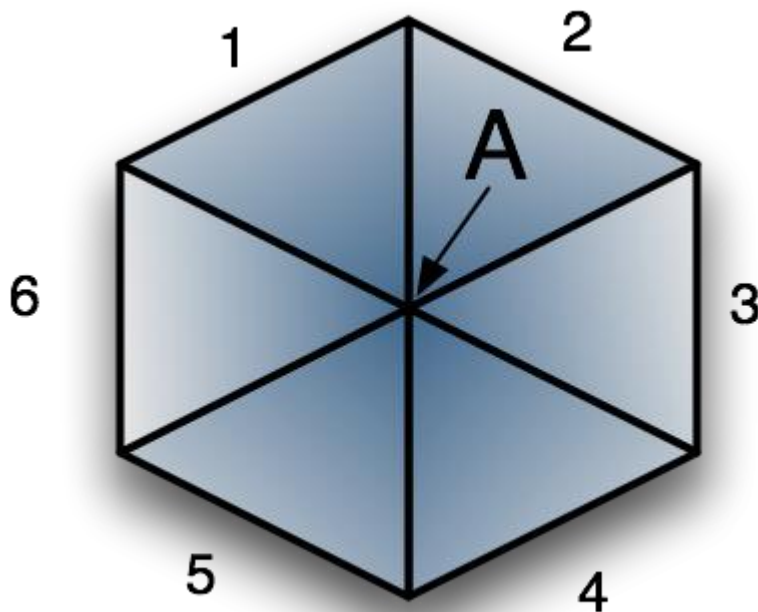
```
static inline Vector3D Triangle3DCalculateSurfaceNormal (Triangle3D
triangle)
{
    Vector3D u = Vector3DMakeWithStartAndEndPoints (triangle.v2,
triangle.v1);
    Vector3D v = Vector3DMakeWithStartAndEndPoints (triangle.v3,
triangle.v1);

    Vector3D ret;
    ret.x = (u.y * v.z) - (u.z * v.y);
    ret.y = (u.z * v.x) - (u.x * v.z);
    ret.z = (u.x * v.y) - (u.y * v.x);
    return ret;
}
```

Vector3DMakeWithStartAndEndPoints() 取两顶点值计算其标准化向量。那么既然计算表面法线如此简单，为什么 OpenGL ES 不为我们完成？有两个原因，第一个和最重要的原因是，开销太大。对每个多边形而言，有许多浮点乘除计算以及调用 sqrtf() 的开销。

第二，因为我们使用 GL_SMOOTH 渲染，所以 OpenGL ES 需要知道**顶点法线 (vertex normal)** 而不是表面法线（上述计算）。因为顶点法线要求你计算使用了该顶点的所有表面法线的平均向量，所以开销更大。

让我们看一个例子。



请注意这不是一个正方体。简单起见，让我们看看一个平面的六个三角形构成的两维形状。它总共由七个顶点构成。顶点 A 由所有六个三角形共享，所以此顶点的顶点法线是所有七个三角形（注：六个吧？）的表面法线的平均值。平均值的计算是基于各向量元素的，即 x 值被平均，y 值被平均，然后 z 值被平均，结果组合在一起构成了平均向量。

所以，我们怎样计算二十面体的向量？这其实是一个很简单的形状，在计算顶点法线时并不会造成显著的延时。通常，你不会工作于这么少顶点的物体，而将处理复杂得多而且数量更多的物体。结果是，除非没有替代方法，否则你希望避免使用顶点法线的实时计算。这种情况下，我编写了一个小命令行程序，它循环处理每个顶点及三角形索引，来计算二十面体的各顶点法线。该程序将结果以 C struct 的形式输出到控制台，然后我复制到我的 OpenGL 程序中。

注意：大部分 3D 程序都会为你提供法线的计算，但你需要小心使用 – 大部分 3D 文件格式存储的是表面法线而不是顶点法线，所以你至少需要计算表面法线的平均值来生成顶点法线。在以后的文章中我将介绍加载和创建 3D 物体，或参阅有关加载 Wavefront OBJ 文件格式的[文章](#)。

下面是我写的计算二十面体顶点法线的命令行程序：

```
#import <Foundation/Foundation.h>
#import "OpenGLCommon.h"
```

```
int main (int argc, const char * argv[]) {
    NSAutoreleasePool * pool = [[NSAutoreleasePool alloc] init];

    NSMutableString *result = [NSMutableString string];

    static const Vertex3D vertices[] = {
```

```

        {0, -0.525731, 0.850651},           // vertices[0]
        {0.850651, 0, 0.525731},           // vertices[1]
        {0.850651, 0, -0.525731},          // vertices[2]
        {-0.850651, 0, -0.525731},         // vertices[3]
        {-0.850651, 0, 0.525731},          // vertices[4]
        {-0.525731, 0.850651, 0},          // vertices[5]
        {0.525731, 0.850651, 0},           // vertices[6]
        {0.525731, -0.850651, 0},          // vertices[7]
        {-0.525731, -0.850651, 0},         // vertices[8]
        {0, -0.525731, -0.850651},         // vertices[9]
        {0, 0.525731, -0.850651},          // vertices[10]
        {0, 0.525731, 0.850651}           // vertices[11]
    };

```

```

static const GLubyte icosahedronFaces[] = {
    1, 2, 6,
    1, 7, 2,
    3, 4, 5,
    4, 3, 8,
    6, 5, 11,
    5, 6, 10,
    9, 10, 2,
    10, 9, 3,
    7, 8, 9,
    8, 7, 0,
    11, 0, 1,
    0, 11, 4,
    6, 2, 10,
    1, 6, 11,
    3, 5, 10,
    5, 4, 11,
    2, 7, 9,
    7, 1, 0,
    3, 9, 8,
    4, 8, 0,
};

```

```

Vector3D *surfaceNormals = calloc(20, sizeof(Vector3D));

```

```

// Calculate the surface normal for each triangle

```

```

for (int i = 0; i < 20; i++)
{

```

```

        Vertex3D vertex1 = vertices[icosahedronFaces[(i*3)]];
        Vertex3D vertex2 = vertices[icosahedronFaces[(i*3)+1]];
        Vertex3D vertex3 = vertices[icosahedronFaces[(i*3)+2]];
        Triangle3D triangle = Triangle3DMake(vertex1, vertex2,
vertex3);
        Vector3D surfaceNormal =
Triangle3DCalculateSurfaceNormal(triangle);
        Vector3DNormalize(&surfaceNormal);
        surfaceNormals[i] = surfaceNormal;
    }

    Vertex3D *normals = calloc(12, sizeof(Vertex3D));
    [result appendString:@"static const Vector3D normals[] = {\n"];
    for (int i = 0; i < 12; i++)
    {
        int faceCount = 0;
        for (int j = 0; j < 20; j++)
        {
            BOOL contains = NO;
            for (int k = 0; k < 3; k++)
            {
                if (icosahedronFaces[(j * 3) + k] == i)
                    contains = YES;
            }
            if (contains)
            {
                faceCount++;
                normals[i] = Vector3DAdd(normals[i],
surfaceNormals[j]);
            }
        }

        normals[i].x /= (GLfloat)faceCount;
        normals[i].y /= (GLfloat)faceCount;
        normals[i].z /= (GLfloat)faceCount;
        [result appendFormat:@"\t%f, %f, %f},\n", normals[i].x,
normals[i].y, normals[i].z];
    }
    [result appendString:@"};\n"];
    NSLog(result);
    [pool drain];
    return 0;
}

```

可能有点粗糙，但它很好的完成了工作，允许我们预先计算顶点法线，所以在运行时不需要进行计算。程序输出如下：

```
static const Vector3D normals[] = {
    {0.000000, -0.417775, 0.675974},
    {0.675973, 0.000000, 0.417775},
    {0.675973, -0.000000, -0.417775},
    {-0.675973, 0.000000, -0.417775},
    {-0.675973, -0.000000, 0.417775},
    {-0.417775, 0.675974, 0.000000},
    {0.417775, 0.675973, -0.000000},
    {0.417775, -0.675974, 0.000000},
    {-0.417775, -0.675974, 0.000000},
    {0.000000, -0.417775, -0.675973},
    {0.000000, 0.417775, -0.675974},
    {0.000000, 0.417775, 0.675973},
};
```

指定顶点法线

首先我们要启动法线数组：

```
glEnableClientState(GL_NORMAL_ARRAY);
```

使用下列调用传递法线数组：

```
glNormalPointer(GL_FLOAT, 0, normals);
```

将所有这些加到 `drawSelf:` 方法中：

```
- (void)drawView:(GLView*)view;
{
```

```
    static GLfloat rot = 0.0;
```

```
    // This is the same result as using Vertex3D, just faster to type and
    // can be made const this way
```

```
    static const Vertex3D vertices[] = {
        {0, -0.525731, 0.850651},           // vertices[0]
        {0.850651, 0, 0.525731},           // vertices[1]
        {0.850651, 0, -0.525731},          // vertices[2]
        {-0.850651, 0, -0.525731},         // vertices[3]
        {-0.850651, 0, 0.525731},          // vertices[4]
        {-0.525731, 0.850651, 0},          // vertices[5]
    };
```

```

        {0.525731, 0.850651, 0},          // vertices[6]
        {0.525731, -0.850651, 0},         // vertices[7]
        {-0.525731, -0.850651, 0},        // vertices[8]
        {0, -0.525731, -0.850651},        // vertices[9]
        {0, 0.525731, -0.850651},         // vertices[10]
        {0, 0.525731, 0.850651}           // vertices[11]
    };

```

```

static const Color3D colors[] = {
    {1.0, 0.0, 0.0, 1.0},
    {1.0, 0.5, 0.0, 1.0},
    {1.0, 1.0, 0.0, 1.0},
    {0.5, 1.0, 0.0, 1.0},
    {0.0, 1.0, 0.0, 1.0},
    {0.0, 1.0, 0.5, 1.0},
    {0.0, 1.0, 1.0, 1.0},
    {0.0, 0.5, 1.0, 1.0},
    {0.0, 0.0, 1.0, 1.0},
    {0.5, 0.0, 1.0, 1.0},
    {1.0, 0.0, 1.0, 1.0},
    {1.0, 0.0, 0.5, 1.0}
};

```

```

static const GLubyte icosahedronFaces[] = {
    1, 2, 6,
    1, 7, 2,
    3, 4, 5,
    4, 3, 8,
    6, 5, 11,
    5, 6, 10,
    9, 10, 2,
    10, 9, 3,
    7, 8, 9,
    8, 7, 0,
    11, 0, 1,
    0, 11, 4,
    6, 2, 10,
    1, 6, 11,
    3, 5, 10,
    5, 4, 11,
    2, 7, 9,
    7, 1, 0,
    3, 9, 8,

```

```

        4, 8, 0,
    };

    static const Vector3D normals[] = {
        {0.000000, -0.417775, 0.675974},
        {0.675973, 0.000000, 0.417775},
        {0.675973, -0.000000, -0.417775},
        {-0.675973, 0.000000, -0.417775},
        {-0.675973, -0.000000, 0.417775},
        {-0.417775, 0.675974, 0.000000},
        {0.417775, 0.675973, -0.000000},
        {0.417775, -0.675974, 0.000000},
        {-0.417775, -0.675974, 0.000000},
        {0.000000, -0.417775, -0.675973},
        {0.000000, 0.417775, -0.675974},
        {0.000000, 0.417775, 0.675973},
    };

    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, -3.0f);
    glRotatef(rot, 1.0f, 1.0f, 1.0f);
    glClearColor(0.7, 0.7, 0.7, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_COLOR_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glColorPointer(4, GL_FLOAT, 0, colors);
    glNormalPointer(GL_FLOAT, 0, normals);
    glDrawElements(GL_TRIANGLES, 60, GL_UNSIGNED_BYTE,
icosahedronFaces);

    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_COLOR_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
    static NSTimeInterval lastDrawTime;
    if (lastDrawTime)
    {
        NSTimeInterval timeSinceLastDraw = [NSDate
timeIntervalSinceReferenceDate] - lastDrawTime;
        rot+=50 * timeSinceLastDraw;
    }
    lastDrawTime = [NSDate timeIntervalSinceReferenceDate];

```



```
}
```

基本完工

运行，你将看到一个真实的三维旋转物体。



但是颜色呢？

请听下回分解：OpenGL ES 材质。当你使用光效和平滑阴影时，OpenGL 期待你为多边形提供材质（**material**）（或纹理（**texture**））。材质比在颜色数组中提供简单颜色要复杂得多。材质像光一样由许多元素构成，可以产生不同的表面效果。物体的表面效果实际上是由场景中的光和多边形的材质决定的。

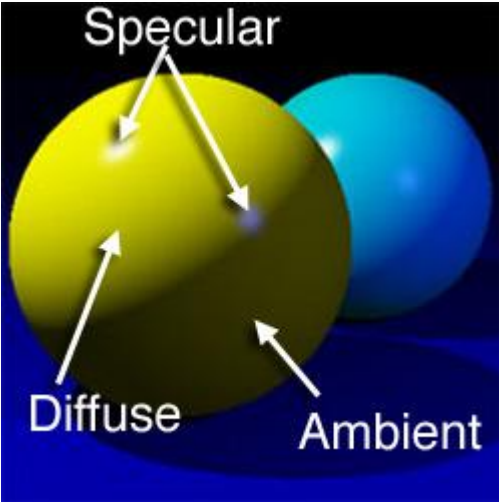
但是，我们不希望显示一个灰暗的二十面体。所以我介绍另一个 OpenGL ES 的配置参数：**GL_COLOR_MATERIAL**。启动它：

```
glEnable(GL_COLOR_MATERIAL);
```

OpenGL 将使用我们提供的颜色数组来创建简单的材质，结果如下：

五. 材质

在 [上一篇文章](#)，我们讨论了光效的设定以及光效的各种属性。我们还讨论了光的三要素：**散射光**、**环境光** 和 **高光**。如果你还不是完全清楚，那么我们来复习一下，在定义材质时大量的用到这些要素。



作为本文的起点，我们使用了[原文中球体绘制](#)的项目文件。我们不再使用二十面体而是转向球体是因为球体是展示光和材质不同要素之间相互作用的最佳形状。

颜色是什么

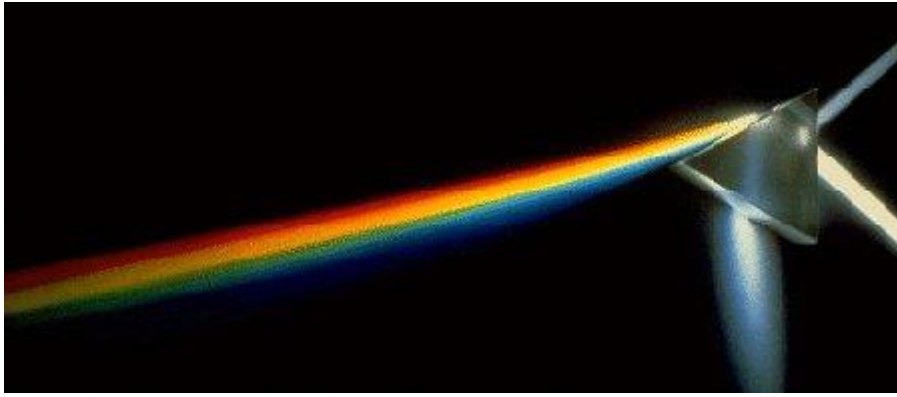
这可能是对小学美术课的复习。为什么现实世界会有颜色？是什么造成的？

我们看得见的光被称为 [光的可见频谱](#)。根据不同的波长我们可以感知到不同的颜色。在可见光谱的一端是低波长高频率的紫色和蓝色，而在另一端是低频高波长的橘色和红色：



电磁波在这个范围之外，因此不是“可见光”，尽管这只是人工的区分方法，它们的唯一不同在频率和波长，在于人眼的感知。尽管有各种方法感知各种电磁波的能力，但从 OpenGL 的角度出发，我们只关心可见光谱。

“白色光”包括等量的所有波长。换言之，白光之所以是白色的是因为它包括了所有（或至少大部分）可见光的频率。如果你曾经做过棱镜试验，那么你可能看过如下效果：



棱镜反射白光，各种波长被分离出来。这就是彩虹产生的原理。

如果你看到一个物体呈蓝色，那么实际上是该物体吸收了大部分可见光谱低频部分。它吸收了红，橘，黄和绿光。根据蓝色的不同色度，它还可能吸收一些紫色和蓝色。

但大部分蓝色的波长都被反射到你的眼睛。因为一些可见光被吸收了，由于它不再包括可见光谱中的所有波长所以反射到你眼中的光不再是白色。

简单吧？让我们看看这些是怎样运用在 OpenGL 的。

OpenGL 材质

我们通过定义材质的反射光来定义 OpenGL ES 中的材质，正如现实世界中一样。如果一个材质定义为反射红光，那么在正常的白光下，它将显示红色。

在 OpenGL 中（至少在使用光滑着色处理和光效时），材质是没有颜色的。OpenGL 具有分别定义材质是怎样反射 OpenGL 光效三要素（环境，散射和高光）的能力。另外，它还具有指定材质 **自发光（emissive）** 属性的能力，关于这点我们稍后再讨论。

指定材质

要在 OpenGL 创建一个材质，我们需要一次或多次调用 `glMaterialf()` 或者 `glMaterialfv()`。类似于上一篇文章中光效的定义，由于各属性或元素需要分别通过这些调用了指定，所以我们通常必须多次调用这些函数以完全定义材质。所有未定义的元素或属性默认值为 0，或者以颜色来说为黑色。

传递给 `glMaterialf()` 或者 `glMaterialfv()` 的第一个参数总是用于指定是否材质影响多边形的前，后或两者的 `GL_ENUM`。实际上除了为了与 OpenGL 兼容，第一个参数在 OpenGL ES 中没有什么意义，因为只有一个有效的选

项：`GL_FRONT_AND_BACK`，它简单地表示材质适用于任何绘制的多边形。如果你还记得[第一部分](#)，那么你应该知道三角形的正面和背面是由 winding（顶点的绘制次序）决定的。默认情况下，只有三角形的正面被绘制出来，但是有可能让 OpenGL 也绘制背面，或甚至只绘制背面，常规 OpenGL 允许你通过传递 `GL_FRONT`，`GL_BACK`，或者 `GL_FRONT_AND_BACK` 来为正面和背面指定不同的材质。但是 OpenGL ES 仅支持 `GL_FRONT_AND_BACK`。

`glMaterialf()` 或 `glMaterialfv()` 的第二个参数是指示正在设定材质的哪个元素或属性的 `GL_ENUM`。它们像传递给 `glLightfv()` 的值一样，比如 `GL_AMBIENT`，另外还有些新的值我们稍后再谈。

最终值是 `GL_FLOAT` 或包括了实际属性或元素的 `GL_FLOAT` 数组的指针。

材质的最重要元素是环境光和散射光，因为它们决定了材质是怎样反射大量光线的。我们今天使用的项目代码定义了正如太阳光或白炽灯产生的白色，它具有平均分布的各种波长和颜色的光。如果光不是白色，球体看上去会有不同的外观。例如，反射至红色材质的蓝光将产生紫色阴影。简单起见，我们只使用白色光。当然，你可以随意改变光的颜色进行试验看看光和材质是怎样交互作用的。大部分时候，它们在 OpenGL ES 中的表示与现实生活中完全一样。

下面是项目在添加材质前运行时的样子：



如你所见，它具有一些环境光和更为显著的散射光。

环境光和散射光

当讨论 OpenGL 的材质时,我们需要同时讨论环境光和散射光,这是因为这两个元素是一起工作从而决定物体被感知的颜色的。记住,散射光处于本文第一个图片中球体的顶部(亮黄色),环境光则是下方的暗黄色。材质怎样反射这两个元素决定了物体被感知的颜色。上图的效果可以通过不止一种方法获得。同样地,黄色球以同样比例反射环境光和散射光,但是场景中具有较少的环境光。

大约 90% 或更多的情况下,将材质的环境光和散射光参数设定成一样。这样做,使它们成为决定物体阴影和外观的因素。实际上,有一种方法通过一个调用 `glMaterialfv()` 同时设定材质环境光和散射光。下面是定义材质为蓝色的示例:

```
GLfloat ambientAndDiffuse[] = {0.0, 0.1, 0.9, 1.0};  
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,  
ambientAndDiffuse);
```

正如 `glColor4f()`, 设置材质指示随后所有物体绘制的方式直到另一个材质被指定。如果将上叙代码放置于我们的绘制代码之前,那么运行时你将看到球体变为蓝色。由于环境光没有散射光强,其下方只是稍暗。



有时你希望更多地控制并希望分别设定材质对环境光和散射光的反射方式。例如,下例中,材质从环境光反射蓝色,从散射光反射红色:

```
GLfloat ambient[] = {0.0, 0.1, 0.9, 1.0};  
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, ambient);  
GLfloat diffuse[] = {0.9, 0.0, 0.1, 1.0};  
glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, diffuse);
```

运行结果：



在此情况下，它看上去像我们投射了有色光到球体上，实际上却不是这样。原因是我们反射了与环境光不一样的定向光线。

大部分情况下，如果你希望产生彩色光的效果，你只需创建彩色光线然后使用 `GL_AMBIENT_AND_DIFFUSE` 来指定材质颜色。但是有时却希望分别设置它们产生特殊效果或在不引起创建额外光线开销的情况下假造一个分离的彩色点光源。记住：你每增加一个光源，也就增加了每秒钟的运算量，所以有时候欺骗并不完全是一件坏事。

高光 and 光泽

你还可以单独设置场景中高光元素的反射方式，从而控制高光“热点”的亮度。一个叫 `GL_SHININESS` 的参数与材质的高光元素一起定义了高光热点的大小。如果你设定了材质的 `GL_SPECULAR` 值，你还应该定义其反光度。反光度越高，高光反射越小，所以默认值 0.0 几乎完全淹没了散射光，因此看上去很糟糕。

让我们回到蓝色球体，增加高光热点：

```
GLfloat ambientAndDiffuse[] = {0.0, 0.1, 0.9, 1.0};  
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE,  
ambientAndDiffuse);  
GLfloat specular[] = {0.3, 0.3, 0.3, 1.0};
```

```
glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, specular);  
glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, 25.0);
```

我们使用了一个较暗的白色作为球体的高光值。这个值看上去有些小，但就是这么小的值也能产生显著的效果。光线高光元素的值与材质的高光元素相乘所产生的光集中在材质反光度指定的区域。下面是上叙代码产生的结果：



现在，球体上有一个小的区域具有更强的反射光。我们可以通过增强光或光的高光元素，或者通过增加材质的反光度使这个点更亮。我们还可以通过调整反光度改变高光的大小。材质反光度越高，高光越集中。例如，如果我们将反光度从 25.0 改为 50.0，我们将得到一个更小的热点，它使得球体显得更具有光泽。



但是，有一点要小心。本文之所以改为使用球体的部分原因以及为什么球体使用较高的顶点数目的原因是高光在一个低面数的物体上看上去实在糟糕。注意一下，如果我减少球体的顶点数会发生什么：



低面片

高光使三角形边缘突出，高光通常在我们的游戏中经常使用的低面片的物体上表现不佳。在常规 OpenGL 中，有一个称为 **着色器 (shader)** 的机制可以用来为低面片物体产生较为理想的结果，但目前 iPhone 上的 OpenGL ES 并不支持此功能（译者注：iPhone 3GS 支持 OpenGL ES 2.0，有 shader 功能。但有一个问题就是 OpenGL ES 1.1 与 OpenGL ES 2.0 并不完全兼容）在游戏中如果你想使低面片物体漂亮的唯一方法就是完全摒弃高光元素而使用纹理映射，这将在下一篇文章中谈到。

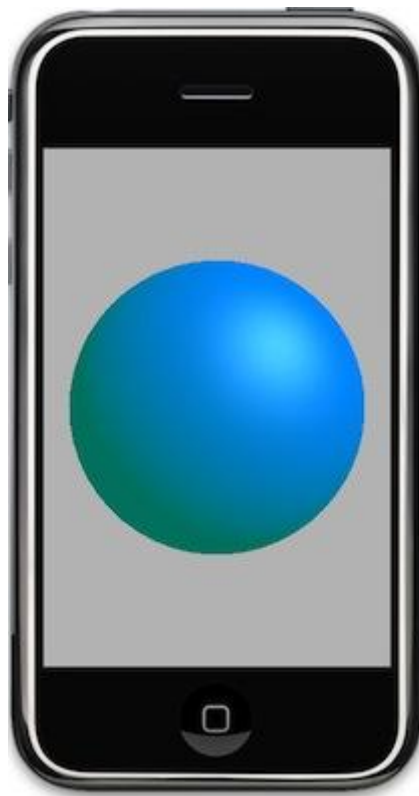
自发光

还剩最后一个材质的重要属性，它称为自发光元素。通过设定自发光元素，使得材质看上去会发射我们指定的颜色。它并不是真正在发光。例如，其周边物体并不会被发射的光线影响。如果你希望一个物体像灯泡一样发光照亮其他物体，由于在 OpenGL ES 中只有光源会发光（听上去像废话），你需要将自发光元素和与物体同一位置处的实际光源结合起来。但是自发光元素可以使物体漂亮地发光。

例如，我们可以为蓝色球体添加绿色的光泽：

```
GLfloat emission[] = {0.0, 0.4, 0.0, 1.0};  
glMaterialfv(GL_FRONT_AND_BACK, GL_EMISSION, emission);
```

结果如下：



自发光元素影响整个材质，因此 `GL_EMISSION` 的值将与落入物体指定区域的任何类型的光相叠加。请注意，甚至上图中的高光部分也成了一点蓝-绿色而不是纯白色。在高光点处其效果是很微小的，但在只有环境光被反射的底部效果更为明显。它实际影响整个物体。



六. 纹理及纹理映射

在 OpenGL ES 中另一种为多边形定义颜色创建材质的方法是将纹理映射到多边形。这是一种很实用的方法，它可以产生很漂亮的外观并节省大量的处理器时间。比如说，你想在游戏中造一个砖墙。你当然可以创建一个具有几千个顶点的复杂物体来定义每块砖以及砖之间的泥灰。

或者你可以创建一个由两个三角形构成的方块（四个顶点），然后将砖的照片映射上去。简单的几何体通过纹理映射的方法比使用材质的复杂几何体的渲染快得多。

功能启动

为使用纹理，我们需要打开 OpenGL 的一些开关以启动我们需要的一些功能：

```
glEnable(GL_TEXTURE_2D);  
glEnable(GL_BLEND);
```

```
glBlendFunc(GL_ONE, GL_SRC_COLOR);
```

第一个函数打开所有两维图像的功能。这个调用是必不可缺的；如果你没有打开此功能，那么你就无法将图像映射到多边形上。它可以在需要时打开和关闭，但是通常不需要这样做。你可以启动此功能而在绘图时并不使用它，所以通常只需在 `setup` 方法中调用一次。

下一个调用打开了**混色 (blending)** 功能。混色提供了通过指定源和目标怎样组合而合成图像的功能。例如，它可以允许你将多个纹理映射到多边形中以产生一个有趣的新的纹理。然而在 OpenGL 中，“混色”是指合成任何图像或图像与多边形表面合成，所以即使你不需要将多个图像混合，你也需要打开此功能。

最后一个调用指定了使用的混色方法。混色函数定义了源图像怎样与目标图像或表面合成。OpenGL 将计算出（根据我们提供的信息）怎样将源纹理的一个像素映射到绘制此像素的目标多边形的一部分。

一旦 OpenGL ES 决定怎样把一个像素从纹理映射到多边形，它将使用指定的混色函数来确定最终绘制的各像素的最终值。`glBlendFunc()` 函数决定我们将怎样进行混色运算，它采用了两个参数。第一个参数定义了怎样使用源纹理。第二个则定义了怎样使用目标颜色或纹理。在本文简单的例子中，我们希望绘制的纹理完全不透明而忽略多边形中现存的颜色或纹理，所以我们设置源为 `GL_ONE`，它表示源图像（被映射的纹理）中各颜色通道的值将乘以 1.0 或者说，以完全颜色密度使用。目标设置为 `GL_SRC_COLOR`，它表示要使用源图像中被映射到多边形特定点的颜色。此混色函数的结果是一个完全不透明的纹理。这可能是最常用情况。我可能会在以后的文章中更详细地介绍一下混色功能，但这可能是你使用最多的一种组合，而且是今天使用的唯一混色功能。

注意：如果你已经使用过 OpenGL 的混色功能，你应该知道 OpenGL ES 并不支持所有 OpenGL 支持的混色功能。

下面是 OpenGL ES 支持的：

```
GL_ZERO, GL_ONE, GL_SRC_COLOR, GL_ONE_MINUS_SRC_COLOR, GL_DST_COLOR, GL_ONE_MINUS_DST_COLOR,  
GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA, GL_DST_ALPHA, GL_ONE_MINUS_DST_ALPHA,  
和 GL_SRC_ALPHA_SATURATE (它仅用于源)。
```

创建纹理

一旦你启动了纹理和混色，就可以开始创建纹理了。通常纹理是在开始显示 3D 物体给用户前程序开始执行时或游戏每关开始加载时创建的。这不是必须的，但却是一个好的建议，因为创建纹理需要占用一些处理器时间，如果你开始显示一些复杂的几何体时进行此项工作，会引起明显的程序停顿。

OpenGL 中的每一个图像都是一个**纹理**，纹理是不能直接显示给最终用户的，除非它**映射**到物体上。但是有一个小小的例外，就是对允许你将图像绘制于指定点的所谓**点精灵 (point sprites)**，但它有自己的一套规则，所以那是一个单独的主题。通常的情况下，任何你希望显示给用户的图像必须放置在由顶点定义的三角形中，有点像贴在上面的粘帖纸。

生成纹理名

为创建一个纹理，首先必须通知 OpenGL ES 生成一个**纹理名称**。这是一个令人迷惑的术语，因为纹理名实际上是一个数字：更具体的说是一个 GLuint。尽管“名称”可以指任何字符串，但对于 OpenGL ES 纹理并不是这样。它是一个代表指定纹理的整数值。每个纹理由一个独一无二的名称表示，所以传递纹理名给 OpenGL 是我们区别所使用纹理的方式。

然而在生成纹理名之前，我们要定义一个保存单个或多个纹理名的 GLuint 数组：

```
GLuint texture[1];
```

尽管只有一个纹理，但使用一个元素的数组而不是一个 GLuint 仍是一个好习惯。当然，仍然可以定义单个 GLuint 进行强制调用。

在过程式程序中，纹理通常存于一个全局数组中，但在 Objective-C 程序中，使用例程变量保存纹理名更为常见。下面是代码：

```
glGenTextures(1, &texture[0]);
```

你可以调用 glGenTextures() 生成多个纹理；传递给 OpenGL ES 的第一个参数指示了要生成几个纹理。第二个参数需要是一个具有足够空间保存纹理名的数组。我们只有一个元素，所以只要求 OpenGL ES 产生一个纹理名。此调用后，texture[0] 将保持纹理的名称，我们将在任何与纹理有关的地方都使用 texture[0] 来表示这个特定纹理。

纹理绑定

在为纹理生成名称后，在为纹理提供图像数据之前，我们必须**绑定**纹理。绑定使得指定纹理处于活动状态。一次只能激活一个纹理。活动的或“被绑定”的纹理是绘制多边形时使用的纹理，也是新纹理数据将加载其上纹理，所以在提供图像数据前必须绑定纹理。这意味着每个纹理至少被绑定一次以为 OpenGL ES 提供此纹理的数据。运行时，可能再次绑定纹理（但不会再次提供图像数据）以指示绘图时要使用此纹理。纹理绑定很简单：

```
glBindTexture(GL_TEXTURE_2D, texture[0]);
```

因为我们使二维图像创建纹理，所以第一个参数永远是 GL_TEXTURE_2D。常规 OpenGL 支持其他类型的纹理，但目前分布在 iPhone 上的 OpenGL ES 版本只支持二维纹理，坦白地说，甚至在常规 OpenGL 中，二维纹理的使用也远比其他类型要多得多。

第二个参数是我们需要绑定的纹理名。调用此函数后，先前生成了纹理名称的纹理将成为活动纹理。

图像配置

在第一次绑定纹理后，我们需要设置两个参数。需要的话，有一些参数**可以**设置，但在 iPhone 上，这两个参数**必须**设定，否则纹理将不会正常显示。

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

必须设置这两个参数的原因是默认状态下 OpenGL 设置了使用所谓 mipmap。今天我将不讨论 mipmap，简单地说，我们不准使用它。Mipmap 是一个图像不同尺寸的组合，它允许 OpenGL 选择最为接近的尺寸版本以避免过多的插值计算并且在物体远离观察者时通过使用更小的纹理来更好地管理内存。感谢矢量单元和图形芯片，iPhone 在图像插值方面做得很好，所以我们不需要考虑 mipmap。我以后可能会专门撰写一篇文章讨论它，但我们今天讨论的是怎样让

OpenGL ES 通过线性插值调整图像到所需的尺寸。因为 `GL_TEXTURE_MIN_FILTER` 用于纹理需要被收缩到适合多边形的尺寸的情形，而 `GL_TEXTURE_MAG_FILTER` 则用于纹理被放大到适合多边形的尺寸的情况下，所以必须进行两次调用。在两种情况下，我们传递 `GL_LINEAR` 以通知 OpenGL 以简单的线性插值方法调整图像。

加载图像数据

在我们第一次绑定纹理后，必须为 OpenGL ES 提供纹理的图像数据。在 iPhone 上，有两种基本方法加载图像数据。如果你在其他书籍上看到使用标准 C I/O 方法加载数据的代码，那也是不错的选择，然而这两种方法应该覆盖了你将遇到的各种情形。

UIImage 方法

如果你想使用 JPEG, PNG 或其他 UIImage 支持的格式，那么你可以简单地使用图像数据实例化一个 UIImage，然后产生图像的 RGBA 位图数据：

```
NSString *path = [[NSBundle mainBundle] pathForResource:@"texture"
ofType:@"png"];
NSData *texData = [[NSData alloc] initWithContentsOfFile:path];
UIImage *image = [[UIImage alloc] initWithData:texData];
if (image == nil)
    NSLog(@"Do real error checking here");

GLuint width = CGImageGetWidth(image.CGImage);
GLuint height = CGImageGetHeight(image.CGImage);
CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
void *imageData = malloc( height * width * 4 );
CGContextRef context = CGContextCreate( imageData, width,
height, 8, 4 * width,
    colorSpace, kCGImageAlphaPremultipliedLast |
kCGBitmapByteOrder32Big );
CGColorSpaceRelease( colorSpace );
CGContextClearRect( context, CGRectMake( 0, 0, width, height ) );
CGContextTranslateCTM( context, 0, height - height );
CGContextDrawImage( context, CGRectMake( 0, 0, width, height ),
image.CGImage );

glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0, GL_RGBA,
    GL_UNSIGNED_BYTE, imageData);

CGContextRelease(context);

free(imageData);
[image release];
```

```
[texData release];
```

前面几行代码很容易理解 – 从程序包中加载一个叫做 texture.png 的图像。然后使用一些 core graphics 调用将位图以 RGBA 格式存放。此基本方法是让我们使用任何 UIImage 支持的图像数据然后转换成 OpenGL ES 接受的数据格式。

注意：只是因为 UIImage 不支持一种文件类型并不意味着你不能使用此方法。你仍然有可能通过使用 Objective-C 的分类来增加 UIImage 对额外的图像文件类型的支持。

一旦具有了正确格式的位图数据，我们就可以调用 `glTexImage2D()` 传递图像数据给 OpenGL ES。完成后，我们释放了一些内存，包括图像数据和实际 UIImage 的实例。一旦你传递图像数据给 OpenGL ES，它就会分配内存以拥有一份自己的数据拷贝，所以你可以释放所有使用的与图像有关的内存，而且你必须这样做除非你的程序有更重要的与数据相关的任务。即使是来自压缩过图像的纹理也会占用程序相当多的内存。每个像素占用四个字节，所以忘记释放纹理图像数据的内存会导致内存很快被用尽。

PVRTC 方法

iPhone 的图形芯片 (PowerVR MBX) 对一种称为 [PVRTC](#) 的压缩技术提供硬件支持，Apple 推荐在开发 iPhone 应用程序时使用 PVRTC 纹理。他们甚至提供了一篇很好的 [技术笔记](#) 描述了怎样通过使用随开发工具安装的命令行程序将标准图像文件转换为 PVRTC 纹理的方法。

你应该知道当使用 PVRTC 时与标准 JPEG 或 PNG 图像相比有可能有些图像质量的下降。是否值得在你的程序中做出一些牺牲取决于一些因素，但使用 PVRTC 纹理可以节省大量的内存空间。

尽管因为没有 Objective-C 类可以解析 PVRTC 数据获取其宽和高¹信息，你想要手工指定图像的高和宽，但加载 PVRTC 数据到当前绑定的纹理实际上甚至比加载普通图像文件更为简单。

下面的例子使用默认的 `texturetool` 设置加载一个 512×512 的 PVRTC 纹理：

```
NSString *path = [[NSBundle mainBundle] pathForResource:@"texture"
ofType:@"pvrtp"];
NSData *texData = [[NSData alloc] initWithContentsOfFile:path];

// This assumes that source PVRTC image is 4 bits per pixel and RGB
not RGBA
// If you use the default settings in texturetool, e.g.:
//
//      texturetool -e PVRTC -o texture.pvrtp texture.png
//
// then this code should work fine for you.
glCompressedTexImage2D(GL_TEXTURE_2D, 0,
GL_COMPRESSED_RGB_PVRTC_4BPPV1_IMG, 512, 512, 0,
[texData length], [texData bytes]);
```

就这么简单。使用 `glCompressedTexImage2D()` 从文件加载数据并传送给 OpenGL ES。而随后怎样处理纹理则绝对没有任何区别。

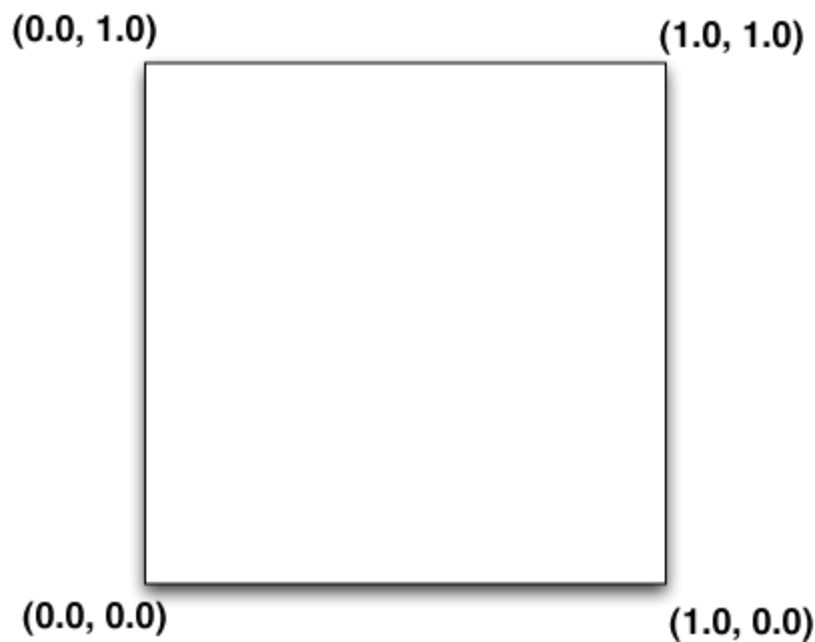
纹理限制

用于纹理的图像宽和高必须为乘方，比如 2, 4, 8, 16, 32, 64, 128, 256, 512, 或 1024。例如图像可能为 64×128 或 512×512 。

当使用 PVRTC 压缩图像时，有一个额外的限制：源图像必须是正方形，所以你的图像应该为 2×2 , 4×4 , 8×8 , 16×16 , 32×32 , 64×64 , 128×128 , 256×256 , 等等。如果你的纹理本身不是正方形，那么你只需为图像加上黑边使图像成为正方形，然后映射纹理使得你需要的部分显示在多边形上。我们现在看看纹理是怎样映射到多边形的。

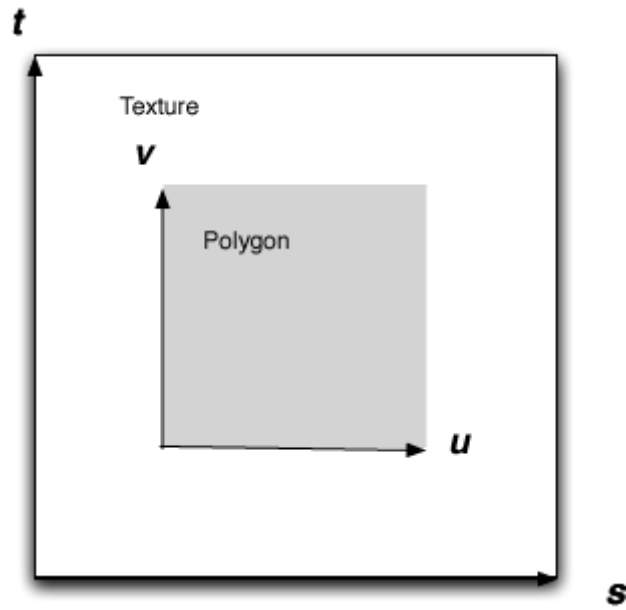
纹理坐标

当纹理映射启动后绘图时，你必须为 OpenGL ES 提供其他数据，即顶点数组中各顶点的 **纹理坐标**。纹理坐标定义了图像的哪一部分将被映射到多边形。它的工作方式有点奇怪。你有一个正方形或长方形的纹理，其左下角为二维平面的原点，高和宽的单位为一。像这样：

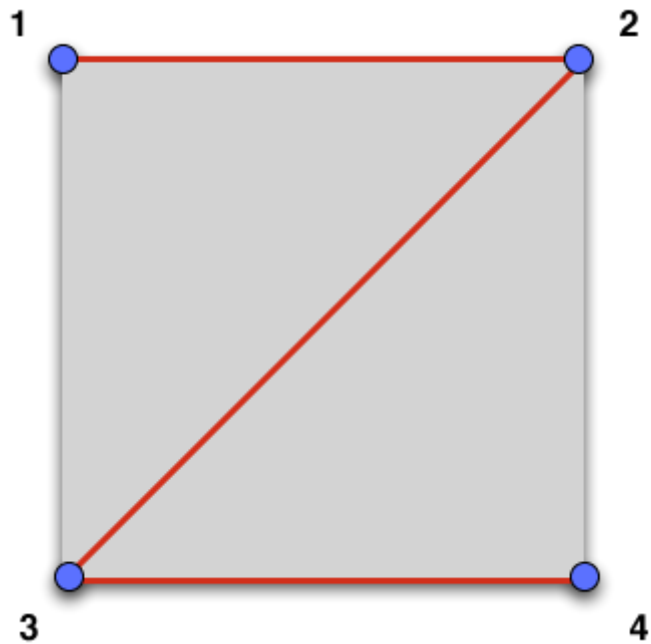


这就是我们的“纹理坐标系统”，不使用 x 和 y 来代表二维空间，我们使用 s 和 t 作为纹理坐标轴，但原理上是一样的。

除了 s 和 t 轴外，被映射的纹理在多边形同样有两个轴，它们称为 u 和 v 轴。这是源于许多 3D 图像程序中的 UV 映射的术语。



好，我们明白了纹理坐标系，我们现在讨论怎样使用这些纹理坐标。当我们指定顶点数组中的顶点时，我们需要在另一个数组中提供纹理坐标，它称为**纹理坐标数组**。每个顶点，我们将传递两个 `GLfloat`s (s, t) 来指定顶点在上图所示坐标系统的位置。让我们看看一个可能是最为简单的例子，将整个图像映射到一个由三角形条组成的正方形上。首先，我们创建一个由四个顶点组成的顶点数组：



现在将两个框图叠在一起，所使用的坐标数组的值变得很明显：

将其转化为 GLfloat 数组：

```
static const GLfloat texCoords[] = {  
    0.0, 1.0,  
    1.0, 1.0,  
    0.0, 0.0,  
    1.0, 0.0  
};
```

为使用纹理坐标数组，我们必须启动它（正如你预料的那样）。使用 `glEnableClientState()`：

```
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

为传递纹理坐标，调用 `glTexCoordPointer()`：

```
glTexCoordPointer(2, GL_FLOAT, 0, texCoords);
```

就是这样。我们汇总一下把代码置于 `drawView:` 方法中。它假设纹理已经被绑定和加载了。

```
- (void)drawView:(GLView*)view;
```

```
{
```

```
    static GLfloat rot = 0.0;
```

```
    glColor4f(0.0, 0.0, 0.0, 0.0);
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
    glEnableClientState(GL_VERTEX_ARRAY);
```

```
    glEnableClientState(GL_NORMAL_ARRAY);
```

```
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

```
    static const Vertex3D vertices[] = {
```

```
        {-1.0, 1.0, -0.0},
```

```

        { 1.0, 1.0, -0.0},
        {-1.0, -1.0, -0.0},
        { 1.0, -1.0, -0.0}
    };
    static const Vector3D normals[] = {
        {0.0, 0.0, 1.0},
        {0.0, 0.0, 1.0},
        {0.0, 0.0, 1.0},
        {0.0, 0.0, 1.0}
    };
    static const GLfloat texCoords[] = {
        0.0, 1.0,
        1.0, 1.0,
        0.0, 0.0,
        1.0, 0.0
    };

    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.0);
    glRotatef(rot, 1.0, 1.0, 1.0);

    glBindTexture(GL_TEXTURE_2D, texture[0]);
    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glNormalPointer(GL_FLOAT, 0, normals);
    glTexCoordPointer(2, GL_FLOAT, 0, texCoords);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);

    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);

    static NSTimeInterval lastDrawTime;
    if (lastDrawTime)
    {
        NSTimeInterval timeSinceLastDraw =
            [NSDate timeIntervalSinceReferenceDate] -
lastDrawTime;
        rot+= 60 * timeSinceLastDraw;
    }
    lastDrawTime = [NSDate timeIntervalSinceReferenceDate];
}

```

下面是我使用的纹理：



运行时的结果：



请等下：那并不正确。如果你仔细对比一下纹理图像和上面的截屏，你就会发现它们并不完全相同。截屏中图像的 y 轴（或 t 轴）完全颠倒了。它上下颠倒了，并不是旋转，而是翻转了。

T-轴翻转之谜

以 OpenGL 的角度来看，我们并未做错任何事情，但结果却是完全错误。原因在于 iPhone 的特殊性。iPhone 中用于 Core Graphics 的图像坐标系统并与 OpenGL ES 一致，其 y 轴在屏幕从上到下而增加。当然在 OpenGL ES 中正好相反，它的 y 轴从下向上增加。其结果就是我们早先传递给 OpenGL ES 中的图像数据从 OpenGL ES 的角度看完全颠倒了。所以，当我们使用标准的 OpenGL ST 映射坐标映射图像时，我们得到了一个翻转的图像。

普通图像的修正

当使用非 PVRTC 图像时，你可以在传递数据到 OpenGL ES 之前就翻转图像的坐标，将下面两行代码到纹理加载中创建 OpenGL 环境的语句之后：

```
CGContextTranslateCTM (context, 0, height);  
CGContextScaleCTM (context, 1.0, -1.0);
```

这将翻转绘制内容的坐标系统，其产生的数据正是 OpenGL ES 所需要的。下面是结果：



PVRTC 图像的修正

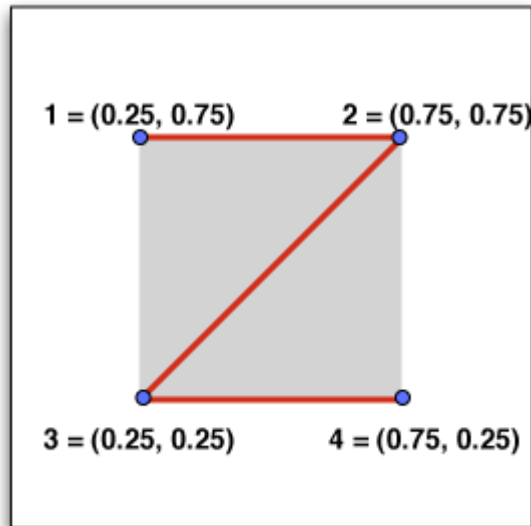
由于没有 UIKit 类可以加载或处理 PVRTC 图像，所以没有一个简单的方法翻转压缩纹理的坐标系统。当然，我们还是有些方法处理这个问题。

一种方法是使用诸如 [Acorn](#) 或 Photoshop 之类的程序中将图像转换为压缩纹理前简单地进行垂直翻转。这看似小计计的方法在很多情况下是最好的解决方法，因为所有的处理都是事前进行的，所以运行时不需要额外的处理时间而且还允许压缩和未压缩图像具有同样的纹理坐标数组。

另一种方法是将 t 轴的值减一。尽管减法是很快，但其占用的时间还是会累积，所以在大部分情况下，尽量避免绘图时进行的转换工作。不论是翻转图像或翻转纹理坐标，都要在显示前进行加载时进行。

更多的映射方式

上个例子中这个图像都被映射到绘制的正方形上。那是因为设定的纹理坐标所决定的。我们可以改变坐标数组仅使用源图像的中心部分。让我们看看仅使用了图像中心部分的另一个框图：



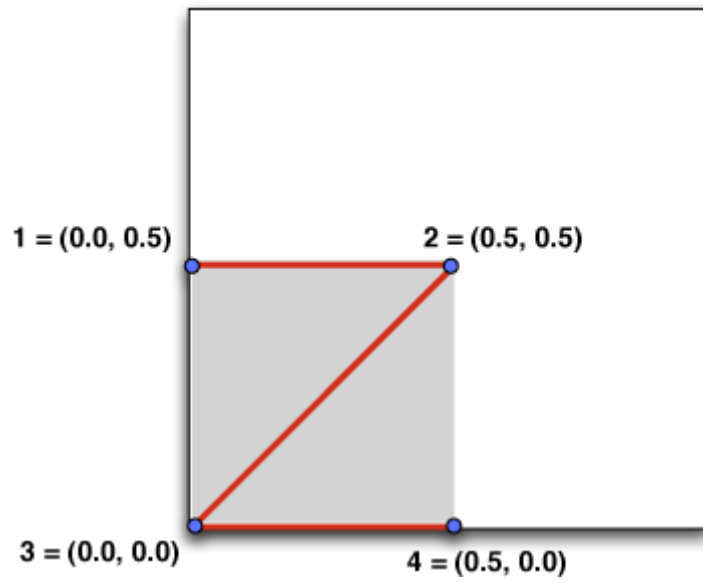
其坐标数组为：

```
static const GLfloat texCoords[] = {  
    0.25, 0.75,  
    0.75, 0.75,  
    0.25, 0.25,  
    0.75, 0.25  
};
```

运行使用了新映射到程序，屏幕上只显示了图像的中心部分：



类似地，如果我们只希望显示纹理的左下部：



坐标数组为:

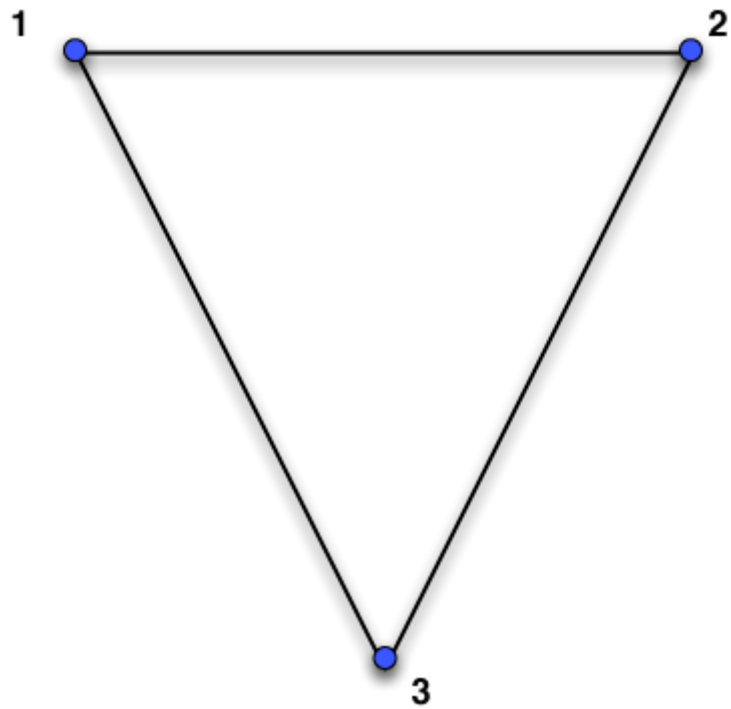
```
static const GLfloat texCoords[] = {  
    0.0, 0.5,  
    0.5, 0.5,  
    0.0, 0.0,  
    0.5, 0.0  
};
```

显示结果：

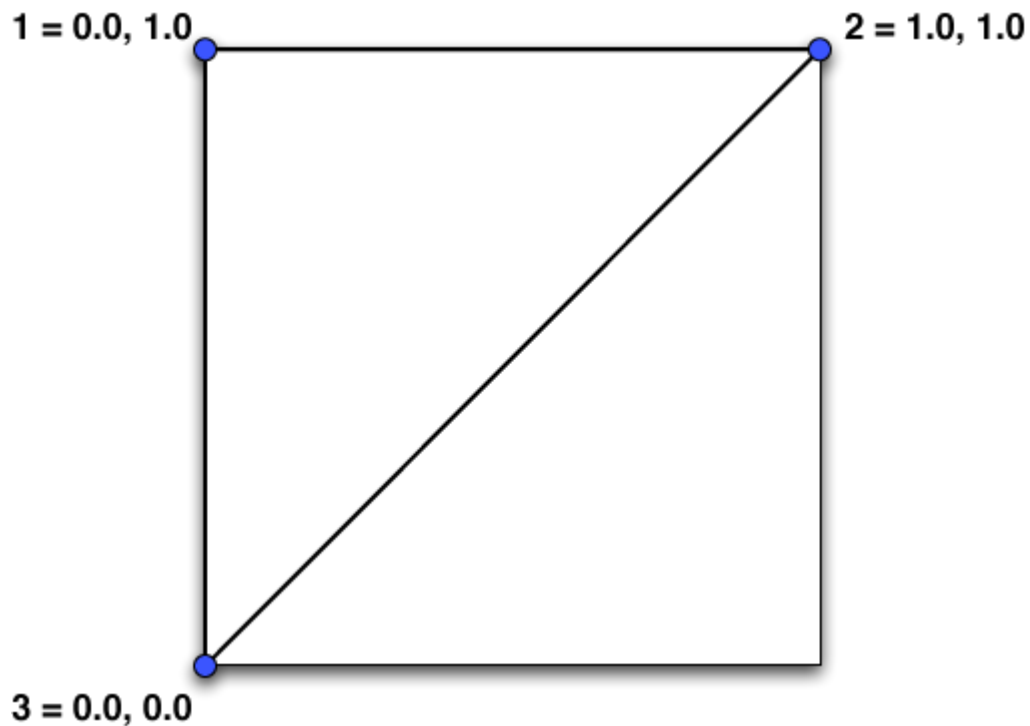


等一下，还有更多的方式

实际上，并不是真正还有更多的映射方式，只是说此功能在正方形到正方形的映射时并不很明显。同样的步骤适合于几何体中任何三角形，而且你甚至可以通过非常规方式的映射来扭曲纹理。例如，我们可以定义一个等腰三角形：



但将底部顶点映射到纹理的左下角：



这样的映射并不会改变几何体 - 它仍然是等腰三角形而不是直角三角形,但 OpenGL ES 将扭曲纹理使得第二个图中的三角形部分以等腰三角形的形式显示出来。代码如下：

```
- (void)drawView:(GLView*)view;
{
    glColor4f(0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);

    static const Vertex3D vertices[] = {
        {-1.0, 1.0, -0.0},
        { 1.0, 1.0, -0.0},
        { 0.0, -1.0, -0.0},
    };

    static const Vector3D normals[] = {
        {0.0, 0.0, 1.0},
        {0.0, 0.0, 1.0},
        {0.0, 0.0, 1.0},
    };

    static const GLfloat texCoords[] = {
```

```

        0.0, 1.0,
        1.0, 0.0,
        0.0, 0.0,
    };

    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.0);

    glBindTexture(GL_TEXTURE_2D, texture[0]);
    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glNormalPointer(GL_FLOAT, 0, normals);
    glTexCoordPointer(2, GL_FLOAT, 0, texCoords);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
}

```

运行时结果如下：



注意到纹理正方形左下角的弧形花纹现在处于三角形的底部了吗？总而言之，纹理上的任何一点都可以映射到多边形的任何一点。或者换而言之，你可以对任何地点 (u, v) 使用任何 (s, t) 而 OpenGL ES 则为你进行映射。

平铺和箝位

我们的纹理坐标系统在两个轴上都是从 0.0 到 1.0，如果设置超出此范围的值会怎么样？根据视图的设置方式有两种选择。

平铺（也叫重复）

一种选择是平铺纹理。按 OpenGL 的术语，也叫“重复”。如果我们将第一个纹理坐标数组的所有 1.0 改为 2.0：

```
static const GLfloat texCoords[] = {  
    0.0, 2.0,  
    2.0, 2.0,  
    0.0, 0.0,  
    2.0, 0.0  
};
```

那么我们得到以下结果：



如果这就是你希望的结果，那么你应该在 `setupView:` 方法中通过 `glTexParameteri()` 函数启动它，像这样：

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

箝位

另一种可能的选择是让 OpenGL ES 简单地将超过 1.0 的值限制为 1.0，任何低于 0.0 的值限制为 0.0。这实际会引起边缘像素重复，从而产生奇怪的效果。下图是使用了箝位的效果：



如果这是你希望的效果，那么你应该在 `setupView:` 方法中使用下面两行代码：

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

注意 `s` 和 `t` 轴是分别设置的，这样有可能在一个方向上使用平铺而在另一个方向使用箝位。

结论

本文介绍了 OpenGL ES 映射纹理到多边形的基本机制。尽管它很简单，但需要动下脑筋并自己动手才能真正理解它到底是怎样工作的，请下载 `texture_projects` 自己测试。

七. 矩阵和变换

今天的主题是我一度谈之色变的。概念上讲，它是 3D 编程中最为困难的部分。

首先，你应该理解 3D 几何和笛卡尔坐标系。你还应该理解由顶点构成的三角形组成的 OpenGL 虚拟世界的物体，各顶点定义了三维空间的特定点，你还应理解怎样使用这些信息在 iPhone 上使用 OpenGL ES 进行绘制。如果你不理解这些概念，我建议你回头再看看我的前六篇文章。

为在交互式程序如游戏中使用这些虚拟世界中的物体，必须要有一种方法来改变物体间的相对位置以及物体与观察者之间的相对位置。要有一种方法不但可以移动，而且可以旋转和改变物体的大小。

还必须要有一种方法将虚拟的三维坐标转换成电脑屏幕的二维坐标。所有这些都是通过所谓变换来实现的。实现变换的内部机制就是矩阵。

尽管你不需要懂得太多有关矩阵和矩阵的数学知识就可以实现许多 OpenGL 的功能，但对这些观念的基本理解有很大的帮助。

内建变换以及单元矩阵

你已经见识许多 OpenGL 的常用变换。其中有一个在每个程序中都可以见到，它就是 `glLoadIdentity()`，我们在 `drawView:` 方法的开始处调用它来进行状态复位。还见识过 `glRotatef()`，用来使二十面体旋转，另外还有 `glTranslatef()` 使物体在虚拟世界中移动。

我们首先看看 `glLoadIdentity()`。此函数加载单元矩阵。我们将稍后讨论此特殊矩阵，但是加载单元矩阵本质上就是将虚拟世界进行复位。它清除了先前应用的任何变换。在绘制开始前调用 `glLoadIdentity()` 是一个很正常的习惯，因为你可以知道起点在什么地方 - 原点，从而可以预料变换的结果。

想知道如果你不调用 `glLoadIdentity()` 会产生什么结果，下载 [第四部分的 Xcode 项目](#)，在 `drawView:` 中为 `glLoadIdentity()` 调用加上注释，运行。发生了什么？

应该慢速旋转的二十面体急速地离开，对吗？像小飞鼠一样飞向天空离开视线。

发生这种现象的原因是在项目中我们使用了两个变换。二十面体的顶点是围绕原点定义的，所以我们使用转移变换将其移动三个单位远离观察者使整个物体可见。我们使用的第二个变换是旋转变换，它使多面体旋转。当使用了 `glLoadIdentity()` 时，每一帧开始时都是回到原点将多面体移离观察者三个单位，也就是说它每次都终止于同一位置 $z = -3.0$ 。类似地，旋转值是随时间增长而增加的，使二十面体以均匀的步调旋转。由于前一个旋转值在开始旋转前被 `glLoadIdentity()` 调用清除掉所以旋转是以匀速进行的。

不调用 `glLoadIdentity()`，尽管第一次二十面体还是移离观察者三个单位并旋转一个小的角度。但从第二帧开始（几十毫秒后），二十面体将继续后移三个单位而且其旋转角度 `rot` 叠加已经旋转的角度。这将发生在每一帧上，这意味着二十面体每帧都会移离三个单位而且旋转的速度都会增加。

尽管我们可以不调用 `glLoadIdentity()` 而对其结果进行补偿，但是由于我们许多时候无法预料转换的结果，所以最好的方法还是让转换开始于已知的位置（通常是原点）并且未经过旋转或尺寸变化，这就是我们总是先调用 `glLoadIdentity()` 的原因了。

常见变换

除了 `glTranslatef()` 和 `glRotatef()` 外，还有 `glScalef()`，它使绘制物体的尺寸增大或减小。在 OpenGL ES 中还有其他一些变换功能，但这三个（与 `glLoadIdentity()` 一起使用）是最常见的。其他功能主要用于将三维虚拟世界转换为二维表示的过程，这个过程称为透视。我们稍后涉及一下透视，但大部分情况下，除了设置视口外我们不需要直接与透视打交道。

这些常用的变换非常实用。你可以仅使用这四个调用就完成整个游戏。但是有时你可能需要自己控制转换。你想要自己控制转换的一个原因是这些常用变换必须按顺序分别调用，每次调用都是一次具有一定开销的矩阵乘法（稍后讨论）。如果你自己进行转换，你可以将多个转换组合成一个矩阵，从而减少每帧都要进行的矩阵乘法操作。

由于你可以向量化你的矩阵乘法调用，所以您可以通过定义自己的矩阵来获取最大性能。据我所知，关于此点 iPhone 并无文档记录，但作为基本规则 OpenGL ES 将对向量间或顶点和矩阵间的乘法进行硬件加速，但两个转换矩阵间的乘法并无加速。通过矩阵乘法向量化，你可以获得比让 OpenGL 进行矩阵乘法更好的性能。由于通常矩阵与矩阵的乘法调用的数量远小于向量/顶点间的矩阵乘法调用的数量，所以这并不会带来巨大的性能提升，但在一个复杂的 3D 程序中，每个方面小的额外性能提升都很有好处。

矩阵

这里我明显不是指电影“黑客帝国”（“The Matrix”），我们将要在随后的篇幅中介绍矩阵。

不幸的是没人可以告诉我矩阵是什么。

实际上，矩阵不过是一个二维数组。就这么简单。没什么神秘的。下面是一个矩阵示例：

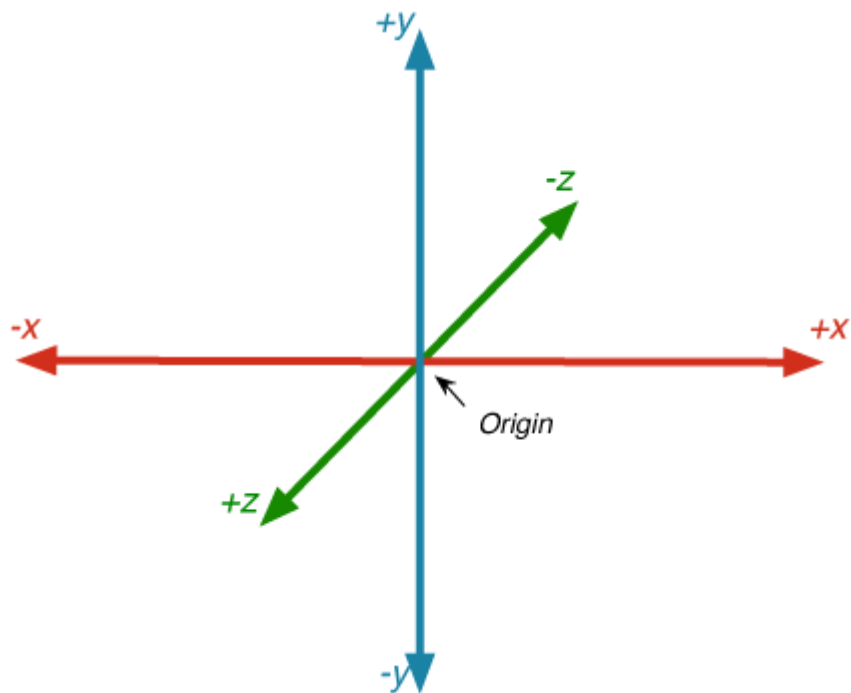
$$\begin{bmatrix} 5 & 23 & -3 \\ -1 & -6 & -11 \\ 18 & 3 & 7 \end{bmatrix}$$

这是一个 3×3 矩阵，它有三行和三列。顶点和向量实际由一个 1×3 的矩阵表示：

$$\begin{bmatrix} x & y & z \end{bmatrix}$$

一个顶点还可以由一个 3×1 数组而不是一个 1×3 数组表示，但是我们这里使用 1×3 格式（后面解释原因）。甚至一个数据元素技术上也可以用 1×1 矩阵表示，尽管这并不是一个十分有用的矩阵。

还有什么可以用数组来表示？坐标系统。还记得向量？向量是想象的从原点指向空间一点的直线。现在，记住笛卡尔坐标系统具有三个轴：



那么，指向 X 轴反向的归一向量是什么样的？记住：归一向量是长度为一的向量，所以一个沿 X 轴正向的归一向量是这样的：

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

注意我们是使用 3×1 矩阵而不是像处理顶点一样使用 1×3 矩阵来表示向量。实际上这并不重要，只要按处理顶点相反的方式处理向量就行。向量中的三个数值适用于同一个坐标轴。我知道，这可能还不太好理解，但我很快就会解释到。Y 轴向上的向量看上去像这样：

$$\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

Z 轴上的向量像这样:

$$\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

现在我们将这三个向量矩阵按照它们在顶点 (x 然后是 y 再是 z) 中的顺序放入矩阵中, 像这样:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

有一个特殊矩阵称为单元矩阵。听起来很熟悉? 当你调用 `glLoadIdentity()` 时, 你就加载了一个单元矩阵¹。这里我解释一下为什么它是一个特殊矩阵。矩阵可以通过相乘而组合在一起。如果你将矩阵乘以一个单元矩阵, 其结果就是原始矩阵, 就像数字乘以一。你可以通过将所有除行号和列号相等的元素外的值设置为 0 来设定一个任意尺寸的单元矩阵, 行号和列号相等处的值为 1.0。

矩阵相乘

矩阵相乘是矩阵组合的关键。如果你有一个定义了转移到矩阵和一个定义了旋转的矩阵, 如果将它们相乘, 你将得到一个既定义了旋转又定义了转移的矩阵。让我们看看一个简单的矩阵相乘的简单例子。看看下面两个矩阵的图像:

$$\begin{bmatrix} 5 & 8 & 1 \\ 6 & 9 & 2 \\ 7 & 3 & 3 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

矩阵相乘的结果是另一个与等式左边矩阵一样尺寸的矩阵。矩阵乘法是不可置换的。顺序是很重要的。将矩阵 a 乘以矩阵 b 的结果不一定与矩阵 b 乘以矩阵 a 的结果相同 (尽管在某些情况下有可能相同)。

有关矩阵相乘还有一件事需要注意：并不是所有的矩阵都可以相乘的。它们并不需要具有同样的尺寸，但等式中右边的矩阵的行数必须与等式中左边矩阵的列数相同。所以，你可以将一个 3×3 的矩阵与另一个 3×3 的矩阵相乘，或者你可以将一个 1×3 的矩阵与一个 3×6 的矩阵相乘，但你不能将一个 2×4 的矩阵与另一个 2×4 的矩阵相乘，因为 2×4 矩阵的列数与另一个 2×4 矩阵的行数并不相同。

要得出矩阵相乘的结果，我们首先建立一个与等式中左边矩阵同样尺寸的空矩阵：

$$\begin{bmatrix} - & - & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

对矩阵中的各点，我们从左边矩阵中取出相应行，再从右边矩阵中取出相应列。所以，对于结果矩阵中的左上方的点，我们取出等式中左边矩阵的第一行以及等式中右边矩阵的第一列，像这样：

$$\begin{bmatrix} 5 & 8 & 1 \\ 6 & 9 & 2 \\ 7 & 3 & 3 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} ? & - & - \\ - & - & - \\ - & - & - \end{bmatrix}$$

然后，将左边矩阵行中第一个值乘以右边矩阵列的第一个值，左边矩阵行第二个值乘以右边矩阵列的第二个值，左边矩阵行第三个值乘以右边矩阵列的第三个值，然后再将它们相加。像这样：

$$(5 \times 1) + (8 \times 0) + (1 \times 0) = 5$$

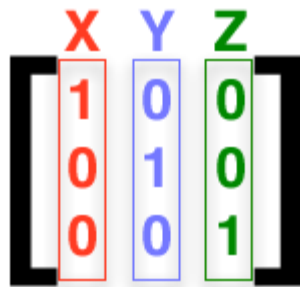
对结果矩阵中的各点重复以上步骤，得到以下结果：

$$\begin{bmatrix} 5 & 8 & 1 \\ 6 & 9 & 2 \\ 7 & 3 & 3 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 8 & 1 \\ 6 & 9 & 2 \\ 7 & 3 & 3 \end{bmatrix}$$

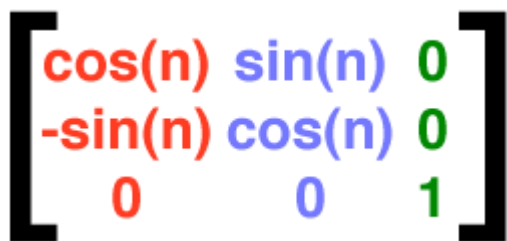
将矩阵（蓝色）乘以单元矩阵（红色），其结果就是原始矩阵。由于单元矩阵代表没有经过任何变换的坐标系统，所以结果完全合理。这也同样适用于顶点。我们将一个顶点乘以一个矩阵：

$$\begin{bmatrix} 5 & 8 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 5 & 8 & 1 \end{bmatrix}$$

现在，我们假定我们希望旋转一个物体。我们要做的就是定义一个描述了被旋转的坐标系统的矩阵。在场景中，我们实际上是旋转了世界坐标，然后将物体绘制其上。如果说我们希望绕 Z 轴旋转一个物体，那么 Z 轴将保持不变，而 X 和 Y 轴将变化。尽管有些不那么直观，要定义一个绕 z 轴旋转的坐标系统，我们需要调整 3×3 矩阵中的 x 和 y 向量，换句话说，我们必须修改第一和第二列。


$$\begin{bmatrix} \text{X} & \text{Y} & \text{Z} \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

所以需要调整通过修改 X 轴向量的 X 值和 Y 向量的 Y 值为旋转角度的余弦值。余弦是三角形角度对应的相邻两边之比。我们还需要修改 X 轴向量的 Y 值为相同角度的正弦值的负值，以及 Y 轴向量的 X 值为相同角度的正弦值。用矩阵来表示可能更容易理解：


$$\begin{bmatrix} \cos(n) & \sin(n) & 0 \\ -\sin(n) & \cos(n) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

现在如果将世界坐标中所有物体的各顶点都乘以这个矩阵，那么物体将被旋转。一旦对物体的所有顶点都进行此操作，那么该物体将沿 Z 轴旋转 n 度。

如果你还不太理解也不要紧。其实你并不需要真正理解使用矩阵的数学原理。它们都是已经解决了的难题，你可以通过 Google 找到任何变换所需的矩阵。实际上，你可以在 OpenGL 的文档中找到大部分。所以如果你不能完全理解为什么这个矩阵导致绕 Z 轴的旋转，也完全不必气馁。

一个 3×3 的矩阵可以描述绕任意轴旋转任何角度的情况。然而，为表示可能遇到的任何变换，我们仍然需要第四行/列。第四列用来保存变换信息，第四行用来表示透视变换。因为需要理解同原坐标以及透视空间，而这些对于成为一个出色的 OpenGL 程序员并不重要，所以我不打算在这里介绍透视变换的数学原理了。要乘以一个 4×4 矩阵，我们需要填充一个附加值，通常我们称其为 W 。 W 应该为 1。在进行乘法运算后，忽略 W 值。我不打算介绍向量的矩阵乘法，因为 OpenGL 已经对此进行了硬件加速，所以通常不需要进行手工处理，但理解其基本步骤是很好的建议。

OpenGL ES 矩阵

OpenGL ES 中有两套矩阵，都是 4×4 的 GLfloat 矩阵。一个叫 modelview matrix，你大部分时间都会与之打交道。它是你用来对虚拟世界进行变换的矩阵。要对虚拟世界中的物体进行旋转，转移或尺寸变化，你都需要对此矩阵进行修改。

另一个矩阵用来创建根据设定的视口对世界坐标进行描述的二维表示。此矩阵称为 projection matrix。在绝大部分时间内，你都不需要接触该矩阵。

任何时刻这两个矩阵中只能有一个是活动的，任何与矩阵相关的调用，包括 glLoadIdentity(), glRotatef(), glTranslatef(), 和 glScalef() 只影响活动矩阵。当你调用 glLoadIdentity 时，活动矩阵设置为单元矩阵。其他三个调用则创建一个转移/尺寸变换/旋转矩阵，并将该矩阵乘以活动矩阵，将活动矩阵的结果替换为矩阵乘法得到的结果。

在大部分情况下，你仅需在程序开始时将 modelview 矩阵设置为活动。实际上，如果你看过我的 OpenGL ES 模板，你将在 setupView: 方法中看到下面代码：

```
glMatrixMode(GL_MODELVIEW);
```

OpenGL ES 的矩阵定义为一个由 16 个 GLfloat 组成的数组，像这样：

```
GLfloat matrix[16];
```

它们也可以表示为一个二维 C 数组，像这样：

```
GLfloat matrix[4][4];
```

两种方法导致同样的内存被分配，所以完全是个人习惯问题，尽管前者更为普遍。

主题

好，我相信现在你已经有了足够的理论基础，想开始进行实践了。首先使用我的模板

[OpenEmpty%20OpenGL%20ES%20Application](#) 创建一个项目，替换 drawView: 和 setupView:

```
– (void)drawView: (GLView*) view;
{
```

```
    static GLfloat  rot = 0.0;
    static GLfloat  scale = 1.0;
    static GLfloat  yPos = 0.0;
    static BOOL      scaleIncreasing = YES;
```

```
    // This is the same result as using Vertex3D, just faster to type and
    // can be made const this way
```

```

static const Vertex3D vertices[] = {
    {0, -0.525731, 0.850651},          // vertices[0]
    {0.850651, 0, 0.525731},          // vertices[1]
    {0.850651, 0, -0.525731},         // vertices[2]
    {-0.850651, 0, -0.525731},        // vertices[3]
    {-0.850651, 0, 0.525731},         // vertices[4]
    {-0.525731, 0.850651, 0},         // vertices[5]
    {0.525731, 0.850651, 0},          // vertices[6]
    {0.525731, -0.850651, 0},         // vertices[7]
    {-0.525731, -0.850651, 0},        // vertices[8]
    {0, -0.525731, -0.850651},        // vertices[9]
    {0, 0.525731, -0.850651},         // vertices[10]
    {0, 0.525731, 0.850651}          // vertices[11]
};

```

```

static const Color3D colors[] = {
    {1.0, 0.0, 0.0, 1.0},
    {1.0, 0.5, 0.0, 1.0},
    {1.0, 1.0, 0.0, 1.0},
    {0.5, 1.0, 0.0, 1.0},
    {0.0, 1.0, 0.0, 1.0},
    {0.0, 1.0, 0.5, 1.0},
    {0.0, 1.0, 1.0, 1.0},
    {0.0, 0.5, 1.0, 1.0},
    {0.0, 0.0, 1.0, 1.0},
    {0.5, 0.0, 1.0, 1.0},
    {1.0, 0.0, 1.0, 1.0},
    {1.0, 0.0, 0.5, 1.0}
};

```

```

static const GLubyte icosahedronFaces[] = {
    1, 2, 6,
    1, 7, 2,
    3, 4, 5,
    4, 3, 8,
    6, 5, 11,
    5, 6, 10,
    9, 10, 2,
    10, 9, 3,
    7, 8, 9,
    8, 7, 0,
    11, 0, 1,
    0, 11, 4,
};

```

```

        6, 2, 10,
        1, 6, 11,
        3, 5, 10,
        5, 4, 11,
        2, 7, 9,
        7, 1, 0,
        3, 9, 8,
        4, 8, 0,
    };

    static const Vector3D normals[] = {
        {0.000000, -0.417775, 0.675974},
        {0.675973, 0.000000, 0.417775},
        {0.675973, -0.000000, -0.417775},
        {-0.675973, 0.000000, -0.417775},
        {-0.675973, -0.000000, 0.417775},
        {-0.417775, 0.675974, 0.000000},
        {0.417775, 0.675973, -0.000000},
        {0.417775, -0.675974, 0.000000},
        {-0.417775, -0.675974, 0.000000},
        {0.000000, -0.417775, -0.675973},
        {0.000000, 0.417775, -0.675974},
        {0.000000, 0.417775, 0.675973},
    };

    glLoadIdentity();
    glTranslatef(0.0f, yPos, -3);
    glRotatef(rot, 1.0f, 1.0f, 1.0f);
    glScalef(scale, scale, scale);

    glClearColor(0.0, 0.0, 0.05, 1.0);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_COLOR_ARRAY);
    glEnable(GL_COLOR_MATERIAL);
    glEnableClientState(GL_NORMAL_ARRAY);
    glVertexPointer(3, GL_FLOAT, 0, vertices);
    glColorPointer(4, GL_FLOAT, 0, colors);
    glNormalPointer(GL_FLOAT, 0, normals);
    glDrawElements(GL_TRIANGLES, 60, GL_UNSIGNED_BYTE,
icosahedronFaces);

    glDisableClientState(GL_VERTEX_ARRAY);

```

```

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_NORMAL_ARRAY);
glDisable(GL_COLOR_MATERIAL);
static NSTimeInterval lastDrawTime;
if (lastDrawTime)
{
    NSTimeInterval timeSinceLastDraw =
        [NSDate timeIntervalSinceReferenceDate] - lastDrawTime;
    rot+=50 * timeSinceLastDraw;

    if (scaleIncreasing)
    {
        scale += timeSinceLastDraw;
        yPos += timeSinceLastDraw;
        if (scale > 2.0)
            scaleIncreasing = NO;
    }
    else
    {
        scale -= timeSinceLastDraw;
        yPos -= timeSinceLastDraw;
        if (scale < 1.0)
            scaleIncreasing = YES;
    }
}
lastDrawTime = [NSDate timeIntervalSinceReferenceDate];
}
-(void)setupView: (GLView*)view
{
    const GLfloat zNear = 0.01, zFar = 1000.0, fieldOfView = 45.0;
    GLfloat size;
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    size = zNear * tanf(DEGREES_TO_RADIANS(fieldOfView) / 2.0);
    CGRect rect = view.bounds;
    glFrustumf(-size, size, -size / (rect.size.width / rect.size.height),
size /
            (rect.size.width / rect.size.height), zNear, zFar);
    glViewport(0, 0, rect.size.width, rect.size.height);
    glMatrixMode(GL_MODELVIEW);

    // Enable lighting

```

```

glEnable(GL_LIGHTING);

// Turn the first light on
glEnable(GL_LIGHT0);

// Define the ambient component of the first light
static const Color3D light0Ambient[] = {{0.3, 0.3, 0.3, 1.0}};
glLightfv(GL_LIGHT0, GL_AMBIENT, (const GLfloat *)light0Ambient);

// Define the diffuse component of the first light
static const Color3D light0Diffuse[] = {{0.4, 0.4, 0.4, 1.0}};
glLightfv(GL_LIGHT0, GL_DIFFUSE, (const GLfloat *)light0Diffuse);

// Define the specular component of the first light
static const Color3D light0Specular[] = {{0.7, 0.7, 0.7, 1.0}};
glLightfv(GL_LIGHT0, GL_SPECULAR, (const GLfloat *)light0Specular);

// Define the position of the first light
// const GLfloat light0Position[] = {10.0, 10.0, 10.0};
static const Vertex3D light0Position[] = {{10.0, 10.0, 10.0}};
glLightfv(GL_LIGHT0, GL_POSITION, (const GLfloat *)light0Position);

// Calculate light vector so it points at the object
static const Vertex3D objectPoint[] = {{0.0, 0.0, -3.0}};
const Vertex3D lightVector =
    Vector3DMakeWithStartAndEndPoints(light0Position[0],
objectPoint[0]);
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, (GLfloat *)&lightVector);

// Define a cutoff angle. This defines a 90° field of vision, since
the cutoff
// is number of degrees to each side of an imaginary line drawn from
the light's
// position along the vector supplied in GL_SPOT_DIRECTION above
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 25.0);

glLoadIdentity();
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
}

```

以上代码使用我们的老朋友 二十面体。与以前一样它旋转，但它同时还使用转移变换沿着 Y 轴上下移动，并且使用尺寸变换矩阵增加和减小尺寸。它使用了所有三个 modelview 变换；我们加载单元矩阵，使用 OpenGL ES 内部变换函数改变尺寸，旋转并且移动。

让我们用自定义的矩阵替换内部函数。在进行修改前，先运行一下看看我们的程序到底应该是怎样工作的。

定义矩阵

我们自定义一个矩阵。

```
typedef GLfloat Matrix3D[16];
```

自定义单元矩阵

首先，我们自定义一个单元矩阵。像这样：

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

下面是代码：

```
static inline void Matrix3DSetIdentity(Matrix3D matrix)
{
    matrix[0] = matrix[5] = matrix[10] = matrix[15] = 1.0;
    matrix[1] = matrix[2] = matrix[3] = matrix[4] = 0.0;
    matrix[6] = matrix[7] = matrix[8] = matrix[9] = 0.0;
    matrix[11] = matrix[12] = matrix[13] = matrix[14] = 0.0;
}
```

第一印象这好像不太对。看上去我们传递的是 Matrix3D 的值。然而我们使用的是 typedef² 数组，由于 C99 数组与指针等价，数组是通过参考而不是值传递的，所以我们只是赋值给数组中的值而不需要直接传递指针。

我使用了内嵌函数以消除函数调用的开销。但是这会有一些副作用（主要是增加了代码的尺寸），本文中的代码与普通 C 函数一样。注意在 C 和 Objective-C 程序中，static 关键字是正确的（而且是很好的方法），但是如果你使用 C++ 或者 Objective-C++，那么你应该移除它。[GCC 手册](#) 推荐在 C 内嵌函数中使用 static，这是因为它允许编译器移除未被使用的内嵌函数所生成的汇编码。然而如果你使用 C++ 或者 Objective-C++，关键字 static 可能会影响链接器的行为而不会提供真正的好处。

好，现在我们使用新函数替代 glLoadIdentity()。删除 glLoadIdentity()然后使用下列代码进行替换：

```
static Matrix3D identityMatrix;
Matrix3DSetIdentity(identityMatrix);
glLoadMatrixf(identityMatrix);
```

我们定义了一个 Matrix3D，赋予其单元矩阵值，然后使用 glLoadMatrixf()加载此矩阵，这使用单元矩阵替代了活动矩阵（即本文情况下的 modelview 矩阵）。这与 glLoadIdentity()调用完全一样。运行，你将看到与以前一样的结果。

现在，你明白了 `glLoadIdentity()` 是怎样工作的。继续。

矩阵乘法

在开始任何更多的变换前，我们需要编写一个两个矩阵相乘的函数。记住矩阵相乘是将两个矩阵合成一个矩阵的方法。我们需要编写一个通用矩阵相乘的方法，它允许任何尺寸的数组并且使用循环进行计算，但是我们最好不要使用循环。循环通常会带来一些开销。由于 OpenGL ES 中的矩阵总是为 4×4 ，最快的方法是分别进行计算。下面是矩阵乘法的代码：

```
static inline void Matrix3DMultiply(Matrix3D m1, Matrix3D m2, Matrix3D result)
{
    result[0] = m1[0] * m2[0] + m1[4] * m2[1] + m1[8] * m2[2] + m1[12]
    * m2[3];
    result[1] = m1[1] * m2[0] + m1[5] * m2[1] + m1[9] * m2[2] + m1[13]
    * m2[3];
    result[2] = m1[2] * m2[0] + m1[6] * m2[1] + m1[10] * m2[2] + m1[14]
    * m2[3];
    result[3] = m1[3] * m2[0] + m1[7] * m2[1] + m1[11] * m2[2] + m1[15]
    * m2[3];

    result[4] = m1[0] * m2[4] + m1[4] * m2[5] + m1[8] * m2[6] + m1[12]
    * m2[7];
    result[5] = m1[1] * m2[4] + m1[5] * m2[5] + m1[9] * m2[6] + m1[13]
    * m2[7];
    result[6] = m1[2] * m2[4] + m1[6] * m2[5] + m1[10] * m2[6] + m1[14]
    * m2[7];
    result[7] = m1[3] * m2[4] + m1[7] * m2[5] + m1[11] * m2[6] + m1[15]
    * m2[7];

    result[8] = m1[0] * m2[8] + m1[4] * m2[9] + m1[8] * m2[10] + m1[12]
    * m2[11];
    result[9] = m1[1] * m2[8] + m1[5] * m2[9] + m1[9] * m2[10] + m1[13]
    * m2[11];
    result[10] = m1[2] * m2[8] + m1[6] * m2[9] + m1[10] * m2[10] + m1[14]
    * m2[11];
    result[11] = m1[3] * m2[8] + m1[7] * m2[9] + m1[11] * m2[10] + m1[15]
    * m2[11];

    result[12] = m1[0] * m2[12] + m1[4] * m2[13] + m1[8] * m2[14] + m1[12]
    * m2[15];
    result[13] = m1[1] * m2[12] + m1[5] * m2[13] + m1[9] * m2[14] + m1[13]
    * m2[15];
```

```

    result[14] = m1[2] * m2[12] + m1[6] * m2[13] + m1[10] * m2[14] + m1[14]
    * m2[15];
    result[15] = m1[3] * m2[12] + m1[7] * m2[13] + m1[11] * m2[14] + m1[15]
    * m2[15];
}

```

此函数不分配任何内存，它只是将两个数组相乘的结果赋予结果数组。结果数组并非两个相乘数组中的任何一个，这是由于如果将结果赋予两个数组中的任意一个，在再次使用时会产生不正确的结果。

但是，等一下... 这实际不是更快一些吗，至少在 iPhone 上如此。iPhone 具有 4 个向量处理器，它们使得进行浮点运算比使用 iPhone CPU 更快。然而使用向量处理器要求编写 ARM6 汇编码因为没有 C 函数库可以访问向量处理器。幸运的是已经有人找到通过使用向量处理器进行矩阵相乘的方法了。[VFP 数学库](#) 包含了许多向量运算功能而且它的许可证要求很宽容。所以我将我们的代码中使用 VFP 数学库进行矩阵相乘，当运行在设备上向量版将被使用而在模拟器上则使用普通版（注意我已经包括了 VFP 数学库的许可证）：

```

/*
    These define the vectorized version of the
    matrix multiply function and are based on the Matrix4Mul method from
    the vfp-math-library. This code has been modified, but is still subject
    to
    the original license terms and ownership as follow:

```

VFP math library for the iPhone / iPod touch

Copyright (c) 2007-2008 Wolfgang Engel and Matthias Grundmann
<http://code.google.com/p/vfpmathlibrary/>

This software is provided 'as-is', without any express or implied warranty.

In no event will the authors be held liable for any damages arising from the use of this software.

Permission is granted to anyone to use this software for any purpose, including commercial applications, and to alter it and redistribute it freely,

subject to the following restrictions:

1. The origin of this software must not be misrepresented; you must not claim that you wrote the original software. If you use this software in a product, an acknowledgment in the product documentation would be appreciated but is not required.
2. Altered source versions must be plainly marked as such, and must not be misrepresented as being the original software.
3. This notice may not be removed or altered from any source distribution.

```

*/
#if TARGET_OS_IPHONE && !TARGET_IPHONE_SIMULATOR
#define VFP_CLOBBER_S0_S31 "s0", "s1", "s2", "s3", "s4", "s5", "s6", "s7",
"s8", \
"s9", "s10", "s11", "s12", "s13", "s14", "s15", "s16", \
"s17", "s18", "s19", "s20", "s21", "s22", "s23", "s24", \
"s25", "s26", "s27", "s28", "s29", "s30", "s31"
#define VFP_VECTOR_LENGTH(VEC_LENGTH) "fmxr    r0, fpscr
\n\t" \
"bic      r0, r0, #0x00370000          \n\t" \
"orr      r0, r0, #0x000" #VEC_LENGTH "0000 \n\t" \
"fmxr     fpscr, r0                    \n\t"
#define VFP_VECTOR_LENGTH_ZERO "fmxr    r0, fpscr          \n\t" \
"bic      r0, r0, #0x00370000 \n\t" \
"fmxr     fpscr, r0                \n\t"
#endif

static inline void Matrix3DMultiply(Matrix3D m1, Matrix3D m2, Matrix3D
result)
{
#if TARGET_OS_IPHONE && !TARGET_IPHONE_SIMULATOR
    __asm__ __volatile__ ( VFP_VECTOR_LENGTH(3)

        // Interleaving loads and adds/muls for faster
calculation.

        // Let A:=src_ptr_1, B:=src_ptr_2, then
        // function computes A*B as (B^T * A^T)^T.

        // Load the whole matrix into memory.
        "fldmias %2, {s8-s23}    \n\t"
        // Load first column to scalar bank.
        "fldmias %1!, {s0-s3}    \n\t"
        // First column times matrix.
        "fmuls s24, s8, s0        \n\t"
        "fmacs s24, s12, s1       \n\t"

        // Load second column to scalar bank.
        "fldmias %1!, {s4-s7}    \n\t"

        "fmacs s24, s16, s2       \n\t"
        "fmacs s24, s20, s3       \n\t"
        // Save first column.
        "fstmias %0!, {s24-s27}  \n\t"

```

```

// Second column times matrix.
"fmuls s28, s8, s4      \n\t"
"fmacs s28, s12, s5     \n\t"

// Load third column to scalar bank.
"fldmias %1!, {s0-s3}   \n\t"

"fmacs s28, s16, s6     \n\t"
"fmacs s28, s20, s7     \n\t"
// Save second column.
"fstmias %0!, {s28-s31} \n\t"

// Third column times matrix.
"fmuls s24, s8, s0      \n\t"
"fmacs s24, s12, s1     \n\t"

// Load fourth column to scalar bank.
"fldmias %1, {s4-s7}    \n\t"

"fmacs s24, s16, s2     \n\t"
"fmacs s24, s20, s3     \n\t"
// Save third column.
"fstmias %0!, {s24-s27} \n\t"

// Fourth column times matrix.
"fmuls s28, s8, s4      \n\t"
"fmacs s28, s12, s5     \n\t"
"fmacs s28, s16, s6     \n\t"
"fmacs s28, s20, s7     \n\t"
// Save fourth column.
"fstmias %0!, {s28-s31} \n\t"

VFP_VECTOR_LENGTH_ZERO
: "=r" (result), "=r" (m2)
: "r" (m1), "0" (result), "1" (m2)
: "r0", "cc", "memory", VFP_CLOBBER_S0_S31
);

```

```

#else

```

```

    result[0] = m1[0] * m2[0] + m1[4] * m2[1] + m1[8] * m2[2] + m1[12]
* m2[3];
    result[1] = m1[1] * m2[0] + m1[5] * m2[1] + m1[9] * m2[2] + m1[13]
* m2[3];

```

```

    result[2] = m1[2] * m2[0] + m1[6] * m2[1] + m1[10] * m2[2] + m1[14]
    * m2[3];
    result[3] = m1[3] * m2[0] + m1[7] * m2[1] + m1[11] * m2[2] + m1[15]
    * m2[3];

    result[4] = m1[0] * m2[4] + m1[4] * m2[5] + m1[8] * m2[6] + m1[12]
    * m2[7];
    result[5] = m1[1] * m2[4] + m1[5] * m2[5] + m1[9] * m2[6] + m1[13]
    * m2[7];
    result[6] = m1[2] * m2[4] + m1[6] * m2[5] + m1[10] * m2[6] + m1[14]
    * m2[7];
    result[7] = m1[3] * m2[4] + m1[7] * m2[5] + m1[11] * m2[6] + m1[15]
    * m2[7];

    result[8] = m1[0] * m2[8] + m1[4] * m2[9] + m1[8] * m2[10] + m1[12]
    * m2[11];
    result[9] = m1[1] * m2[8] + m1[5] * m2[9] + m1[9] * m2[10] + m1[13]
    * m2[11];
    result[10] = m1[2] * m2[8] + m1[6] * m2[9] + m1[10] * m2[10] + m1[14]
    * m2[11];
    result[11] = m1[3] * m2[8] + m1[7] * m2[9] + m1[11] * m2[10] + m1[15]
    * m2[11];

    result[12] = m1[0] * m2[12] + m1[4] * m2[13] + m1[8] * m2[14] + m1[12]
    * m2[15];
    result[13] = m1[1] * m2[12] + m1[5] * m2[13] + m1[9] * m2[14] + m1[13]
    * m2[15];
    result[14] = m1[2] * m2[12] + m1[6] * m2[13] + m1[10] * m2[14] + m1[14]
    * m2[15];
    result[15] = m1[3] * m2[12] + m1[7] * m2[13] + m1[11] * m2[14] + m1[15]
    * m2[15];
#endif
}

```

既然我们有将矩阵相乘的能力，我们就可以合并多个矩阵。由于我们的矩阵相乘的方法是硬件加速的，而 OpenGL ES 不支持矩阵与矩阵相乘的硬件加速，所以我们的方法应该比内嵌的变换方法要快一些³。我们现在开始进行转移变换。

自定义转移

如果你还记得，我们使用 4×4 矩阵而不是使用 3×3 矩阵的一个原因是我们需要额外的一列用来保存转移信息。转移矩阵看上去像这样：

$$\begin{bmatrix} 1 & 0 & 0 & \Delta x \\ 0 & 1 & 0 & \Delta y \\ 0 & 0 & 1 & \Delta z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

我们可以将其转换为函数：

```
static inline void Matrix3DSetTranslation(Matrix3D matrix, GLfloat xTranslate,
    GLfloat yTranslate, GLfloat zTranslate)
{
    matrix[0] = matrix[5] = matrix[10] = matrix[15] = 1.0;
    matrix[1] = matrix[2] = matrix[3] = matrix[4] = 0.0;
    matrix[6] = matrix[7] = matrix[8] = matrix[9] = 0.0;
    matrix[11] = 0.0;
    matrix[12] = xTranslate;
    matrix[13] = yTranslate;
    matrix[14] = zTranslate;
}
```

现在，我们怎样将其引入到 `drawView` 方法中？我们可以删除 `glTranslatef()`，替换为定义另一个矩阵并赋予适当的平移值，然后将其与当前矩阵相乘最后将结果加载到 OpenGL 的代码，对吗？

```
static Matrix3D identityMatrix;
Matrix3DSetIdentity(identityMatrix);
static Matrix3D translateMatrix;
Matrix3DSetTranslation(translateMatrix, 0.0, yPos, -3.0);
static Matrix3D resultMatrix;
Matrix3DMultiply(identityMatrix, translateMatrix, resultMatrix);
glLoadMatrixf(resultMatrix);
```

是的，这可以工作，但它做了一些不必要的工作。记住，如果你将矩阵与单元矩阵相乘，其结果一定是矩阵本身。所以当使用自定义矩阵时，如果要进行任何变换，我们不再需要首先加载单元矩阵。我们只需要创建变换矩阵并加载它：

```
static Matrix3D translateMatrix;
Matrix3DSetTranslation(translateMatrix, 0.0, yPos, -3.0);
glLoadMatrixf(translateMatrix);
```

由于不需要加载单元矩阵，此方法可以省去一些工作。另外，注意我将 `Matrix3D` 定义为 `static`。我们不希望经常分配和解除内存分配。我们知道当程序运行时每一秒钟都需要用到此矩阵许多次，所以将其定义为 `static` 可以省去分配和解除内存分配的开销。

自定义尺寸变换

一个用于物体尺寸变换的矩阵像这样：

$$\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

x , y , 或 z 为 1.0 表示在相应方向上尺寸无变化。3 个值都为 1.0 代表单元矩阵。如果你赋值为 2.0，则物在相应轴上尺寸加倍。我们可以将尺寸变换矩阵转化为 OpenGL ES 矩阵，像这样：

```
static inline void Matrix3DSetScaling(Matrix3D matrix, GLfloat xScale,
    GLfloat yScale, GLfloat zScale)
{
    matrix[1] = matrix[2] = matrix[3] = matrix[4] = 0.0;
    matrix[6] = matrix[7] = matrix[8] = matrix[9] = 0.0;
    matrix[11] = matrix[12] = matrix[13] = matrix[14] = 0.0;
    matrix[0] = xScale;
    matrix[5] = yScale;
    matrix[10] = zScale;
    matrix[15] = 1.0;
}
```

现在，因为我们需要使用超过一种变换，我们将使用矩阵乘法。要进行尺寸变换和旋转，我们需要将这两个矩阵相乘。删除先前代码中的 `glScalef()` 替换为下列代码：

```
static Matrix3D    translateMatrix;
Matrix3DSetTranslation(translateMatrix, 0.0, yPos, -3.0);
static Matrix3D    scaleMatrix;
Matrix3DSetScaling(scaleMatrix, scale, scale, scale);
static Matrix3D    resultMatrix;
Matrix3DMultiply(translateMatrix, scaleMatrix, resultMatrix);
glLoadMatrixf(resultMatrix);
```

我们创建一个矩阵并赋予其适当的转移值。然后创建尺寸变换矩阵并赋值。然后将这两个矩阵相乘将结果加载到模型视图矩阵中。

八. 交叉存取定点数据

Technote 2230 提出了很多用 OpenGL ES 来提升 iPhone 程序性能的建议。我们现在远远不能深刻理解 OpenGL ES 所以

你需要学习以下内容。不信？是真的，试试看，我等着你的读后感。

好，就这样定了？副标题为“优化顶点数据”，这里有一些算法上的建议用来“**submit strip-ordered indexed triangles with per vertex data interleaved**”。当苹果给出这些建议的时候，他们通常有一些非常好的理由，让我们看看如何使用它。首先让我们了解它的意思。让我们把这句话分解开来：

Strip Ordered: 换句话说，如果你的模型有相邻的三角形，组成一个三角形带提交而不是分别提交每一个三角形。我们早期教程讲到过三角形带，你应该多少知道一点该怎么做。对于大多数物体来说你都可以使用这个方法，但不是所有时候都要用它。你什么时候能够使用？当你确定使用三角形带可以有效减少推入 OpenGL ES 每帧的顶点数据的时候。

indexed: 这里仍然没有什么新内容。我们使用顶点索引有一段时间了。我们仅仅用 12 个顶点来创建旋转的 20 面体。`glDrawElements()`是基于索引绘制而不是基于顶点。

见鬼，我们目前已经做的很棒了，不是吗？我们眼下一直在做着正确的事情。让我们来看最后一个建议，然而：

with per vertex data interleaved: 好的，恩。。。它到底是什么意思？

好的，是时候考验你的记忆力了。你还记得在过去几个部分，也就是当我们讨论 `glVertexPointer()`, `glNormalPointer()`, `glColorPointer()`, `glTexCoordPointer()` 的时候吗？在前面的部分里，我告诉你不用关注参数 `stride` 并把它设置为 0。

但是现在你要关注它（`stride`）了，因为它是用来交叉存取每个顶点数据的关键。

Per Vertex Data

你也许想知道“**per vertex data**”是什么，并且想知道如何交叉存取它。

你肯定知道，在 OpenGL ES 里我们用顶点数组来表示几何图形，每一个数组包含 3 个定义顶点的 `GLfloat` 变量，来创造我们的物体。除了这些，我们有时也使用其它数据。举例来说，如果我们用光效就需要顶点法线，我们不得不在法线数组里定义每一个顶点的法线。如果我们使用纹理坐标，我们不得不在纹理数组里定义每一个顶点的纹理坐标。如果我们用颜色数组，我们不得不指定每个点的颜色，你是否注意到我总是强调“**per vertex data**”？这些数据类型就是苹果公司在它们技术说明里提到的“**per vertex data**”。这就是你所有在 OpenGL ES 中放入数组的任意一种适用于顶点的顶点数组。

Interleaving

这个系列到目前为止，我们已经创造一个数组来存放顶点数据，并把它们与法线数据、颜色数据、纹理坐标分开存在其他独立数组中，像这样：


```

vertices
static const Vertex3D vertices[] = {
    {0, -0.525731, 0.850651},
    {0.850651, 0, 0.525731},
    {0.850651, 0, -0.525731},
    {-0.850651, 0, -0.525731},
    {-0.850651, 0, 0.525731},
    {-0.525731, 0.850651, 0},
    {0.525731, 0.850651, 0},
    {0.525731, -0.850651, 0},
    {-0.525731, -0.850651, 0},
    {0, -0.525731, -0.850651},
    {0, 0.525731, -0.850651},
    {0, 0.525731, 0.850651}
};

```

```

normals
static const Vector3D normals[] = {
    {0.000000, -0.417775, 0.675974},
    {0.675973, 0.000000, 0.417775},
    {0.675973, -0.000000, -0.417775},
    {-0.675973, 0.000000, -0.417775},
    {-0.675973, -0.000000, 0.417775},
    {-0.417775, 0.675974, 0.000000},
    {0.417775, 0.675973, -0.000000},
    {0.417775, -0.675974, 0.000000},
    {-0.417775, -0.675974, 0.000000},
    {0.000000, -0.417775, -0.675973},
    {0.000000, 0.417775, -0.675974},
    {0.000000, 0.417775, 0.675973},
};

```

```

colors
static const Color3D colors[] = {
    {1.0, 0.0, 0.0, 1.0},
    {1.0, 0.5, 0.0, 1.0},
    {1.0, 1.0, 0.0, 1.0},
    {0.5, 1.0, 0.0, 1.0},
    {0.0, 1.0, 0.0, 1.0},
    {0.0, 1.0, 0.5, 1.0},
    {0.0, 1.0, 1.0, 1.0},
    {0.0, 0.5, 1.0, 1.0},
    {0.0, 0.0, 1.0, 1.0},
    {0.5, 0.0, 1.0, 1.0},
    {1.0, 0.0, 1.0, 1.0},
    {1.0, 0.0, 0.5, 1.0}
};

```

我们将要学习如何把这所有的数据放在一起作为一个整体数据来存储：

vertexData

```
static const ColoredVertexData3D vertexData[] = {
    {
        {0, -0.525731, 0.850651},           // Vertex |
        {0.000000, -0.417775, 0.675974},    // Normal | Vertex 0
        {1.0, 0.0, 0.0, 1.0}               // Color |
    },
    {
        {0.850651, 0, 0.525731},           // Vertex |
        {0.675973, 0.000000, 0.417775},    // Normal | Vertex 1
        {1.0, 0.5, 0.0, 1.0}               // Color |
    },
    {
        {0.850651, 0, -0.525731},          // Vertex |
        {0.675973, -0.000000, -0.417775},  // Normal | Vertex 2
        {1.0, 1.0, 0.0, 1.0}               // Color |
    },
    {
        {-0.850651, 0, -0.525731},         // Vertex |
        {-0.675973, 0.000000, -0.417775},  // Normal | Vertex 3
        {0.5, 1.0, 0.0, 1.0}               // Color |
    },
    {
        {-0.850651, 0, 0.525731},          // Vertex |
        {-0.675973, -0.000000, 0.417775},  // Normal | Vertex 4
        {0.0, 1.0, 0.0, 1.0}               // Color |
    },
    {
        {-0.525731, 0.850651, 0},          // Vertex |
        {-0.417775, 0.675974, 0.000000},  // Normal | Vertex 5
        {0.0, 1.0, 0.5, 1.0}               // Color |
    },
    {
        {0.525731, 0.850651, 0},           // Vertex |
        {0.417775, 0.675973, -0.000000},  // Normal | Vertex 6
        {0.0, 1.0, 1.0, 1.0}               // Color |
    },
    {
        {0.525731, -0.850651, 0},          // Vertex |
        {0.417775, -0.675974, 0.000000},  // Normal | Vertex 7
        {0.0, 0.5, 1.0, 1.0}               // Color |
    },
    {
        {-0.525731, -0.850651, 0},         // Vertex |
        {-0.417775, -0.675974, 0.000000},  // Normal | Vertex 8
        {0.0, 0.0, 1.0, 1.0}               // Color |
    },
    {
        {0, -0.525731, -0.850651},         // Vertex |
        {0.000000, -0.417775, -0.675973},  // Normal | Vertex 9
        {0.5, 0.0, 1.0, 1.0}               // Color |
    },
    {
        {0, 0.525731, -0.850651},          // Vertex |
        {0.000000, 0.417775, -0.675974},  // Normal | Vertex 10
        {1.0, 0.0, 1.0, 1.0}               // Color |
    },
    {
        {0, 0.525731, 0.850651},           // Vertex |
        {0.000000, 0.417775, 0.675973},    // Normal | Vertex 11
        {1.0, 0.0, 0.5, 1.0}               // Color |
    }
};
```

如果你读不懂上图解里面的代码请不要担心。 当那个变得重要的时候，我们会再次列出来讲解的，这个给出的代码列表仅仅是举例说明在一个独立的内存单元中我们可以存入所有的顶点数据。 我们所要做的只是把所有描述一个单一点的数据放在内存的同一个地方。

这样做能够使 OpenGL 快速的获得读取到每个顶点的信息。 在今天的例子里面，我们要将交叉存储顶点，法线，颜色（vertices, normals, color data），同样的方法可以使用在纹理坐标中，或者仅交叉存储顶点和法线。事实上在一个 Xcode

工程里面，会附带着一些数据结构定义这三个交叉存储的情况。

Defining a Vertex Node

为了让这个能够工作起来，我们需要一个新的数据结构。下面的数据结构能够让我们把上面提到的顶点，法线，颜色 交叉存储到一起，

```
typedef struct {  
    Vertex3D    vertex;  
    Vector3D    normal;  
    Color3D     color;  
} ColoredVertexData3D;
```

啊哈哈，漂亮而简洁不是吗？你只用一个数据结构就包含了我们需要的一个顶点的所有属性。

下一步，当然了，线面我们需要填充顶点数据，所以我们需要把三个 **static const** 数组合并成一个。这里是相同的二十面体数据，指定使用了我们自己定义的新数据类型：

```
static const ColoredVertexData3D vertexData[] = {  
    {  
        {0, -0.525731, 0.850651},          // Vertex |  
        {0.000000, -0.417775, 0.675974},    // Normal  | Vertex 0  
        {1.0, 0.0, 0.0, 1.0}                // Color   |  
    },  
    {  
        {0.850651, 0, 0.525731},            // Vertex |  
        {0.675973, 0.000000, 0.417775},      // Normal  | Vertex 1  
        {1.0, 0.5, 0.0, 1.0}                // Color   |  
    },  
    {  
        {0.850651, 0, -0.525731},           // Vertex |  
        {0.675973, -0.000000, -0.417775},    // Normal  | Vertex 2  
        {1.0, 1.0, 0.0, 1.0}                // Color   |  
    },  
    {  
        {-0.850651, 0, -0.525731},          // Vertex |  
        {-0.675973, 0.000000, -0.417775},    // Normal  | Vertex 3  
        {0.5, 1.0, 0.0, 1.0}                // Color   |  
    },  
    {  
        {-0.850651, 0, 0.525731},           // Vertex |
```

```

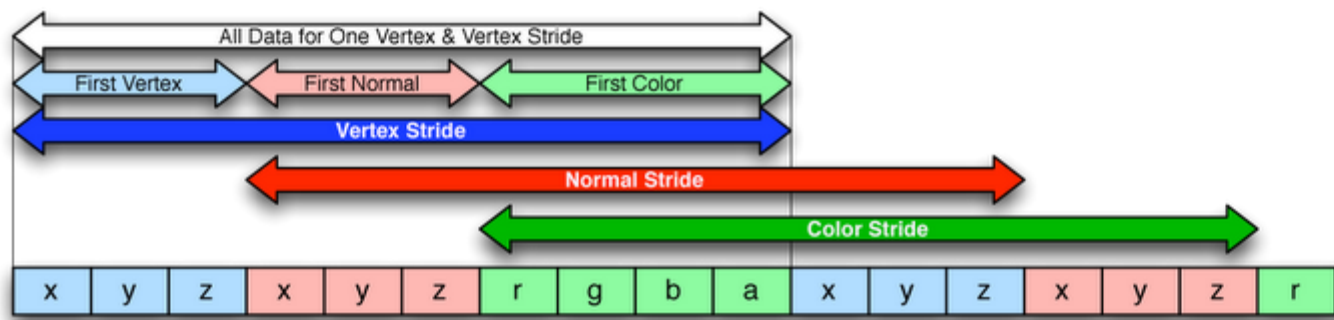
        {-0.675973, -0.000000, 0.417775}, // Normal | Vertex 4
        {0.0, 1.0, 0.0, 1.0} // Color |
    },
    {
        {-0.525731, 0.850651, 0}, // Vertex |
        {-0.417775, 0.675974, 0.000000}, // Normal | Vertex 5
        {0.0, 1.0, 0.5, 1.0} // Color |
    },
    {
        {0.525731, 0.850651, 0}, // Vertex |
        {0.417775, 0.675973, -0.000000}, // Normal | Vertex 6
        {0.0, 1.0, 1.0, 1.0} // Color |
    },
    {
        {0.525731, -0.850651, 0}, // Vertex |
        {0.417775, -0.675974, 0.000000}, // Normal | Vertex 7
        {0.0, 0.5, 1.0, 1.0} // Color |
    },
    {
        {-0.525731, -0.850651, 0}, // Vertex |
        {-0.417775, -0.675974, 0.000000}, // Normal | Vertex 8
        {0.0, 0.0, 1.0, 1.0}, // Color |
    },
    {
        {0, -0.525731, -0.850651}, // Vertex |
        {0.000000, -0.417775, -0.675973}, // Normal | Vertex 9
        {0.5, 0.0, 1.0, 1.0} // Color |
    },
    {
        {0, 0.525731, -0.850651}, // Vertex |
        {0.000000, 0.417775, -0.675974}, // Normal | Vertex 10
        {1.0, 0.0, 1.0, 1.0} // Color |
    },
    {
        {0, 0.525731, 0.850651}, // Vertex |
        {0.000000, 0.417775, 0.675973}, // Normal | Vertex 11
        {1.0, 0.0, 0.5, 1.0} // Color |
    }
};

```

下面是我们如何传递信息到 **OpenGL**。我们传递在这个数组里面的每个数据成员的第一个顶点的地址，并且提供所传递的数据的长度大小作为步长参数，而不是传递指针到合适的数组。

```
glVertexPointer(3, GL_FLOAT, sizeof(ColoredVertexData3D), &vertexData[0].vertex);
glColorPointer(4, GL_FLOAT, sizeof(ColoredVertexData3D), &vertexData[0].color);
glNormalPointer(GL_FLOAT, sizeof(ColoredVertexData3D), &vertexData[0].normal);
```

以上几个方法中，最后一个参数调用了数据里的第一个顶点，例如，`&vertexData[0].color` 指向第一个顶点的颜色信息。
stride（跨度）参数表明了在我们读取下一个相同类型的数据时，我们要跳过多少比特的数据。如果你看了下面的图解你也许就有那么一点点感觉了（不好意思它有点宽，你需要适当的调整你的浏览器以便于看到这个图解的所有部分）



变简单了不是吗？如果你一点也不喜欢打字，那么你可以下载这个十二面的交叉存取版本 [spinning icosahedron](#)。我也会使用新的数据类型更新我的 [OpenGL ES Xcode Template](#)。

我们现在依然没有使用 **triangle strips**，但是混合三角形到 **triangle strips** 是下一步的教程，现在我要去参加 WWDC 了。

九. 动画基础和关键帧动画

最初这篇教程我本打算作为第 9 章发布，原计划是第 10 章。在深入了解 OpenGL ES 2.0 和着色器之前，我想讨论下更基础的：动画。

注意：你可以在这里找到这篇教程的配套[代码](#)，新版本的代码已经在西部时间 10: 14 更新了，更新的代码里面修正了一个不能动画的错误。

目前为止，想必你已经看过了 **opengles** 最基本的动画形式。通过随时间改变 **rotate**, **translate**, **scale**（旋转、移动和缩放）等，我们就可以使物体“动起来”。我们的第一个项目 **the spinning icosahedron** 就是这种动画的一个例子。我们把这种动画叫做简单动画。然而，不要被“简单动画”这个名称迷糊，你可以实现复杂的动画，只需要随时间改变一下矩阵变换。

但是，如何掌握更加复杂的动画呢？比如说你想让一个人物行走或者表现一个被挤压正要反弹的球。

实际上这并不困难。在 **OpenGL** 了里面有两种主要实现方法：关键帧动画和骨骼动画。在这章里面我们谈论关于帧动画的话题，下一章([#9b](#))里面，我们将要谈论的是骨骼动画。

Interpolation & Keys

动画只不过是随着时间改变每个顶点的位置。这是动画的本质。当你移动、旋转或缩放一个物体的时候，你实际上是移动了一个物体的所有顶点。如果你想让一个物体有一个更复杂、精细的动画，你需要一个方法按设置时间移动每个顶点。

两种动画的基本原理是存储物体关键位置的每一个顶点。在关键帧动画中，我们存储独立关键位置的每一个顶点。而骨骼动画，我们存储虚拟骨骼的位置信息，并且用一些方法指定哪个骨骼会影响动作中的哪些顶点。

那么什么是关键帧？如果要最简单的方法说明他们，我们还得回到他们的起源，传统逐格动画，如经典的迪斯尼和华纳兄弟的卡通。早期的动画，一个小的团队就能完成所有的绘画工作。但是随着产品的慢慢变大，那变得不可能，他们不得不进行分工。比较有经验的漫画师成为 **lead animator**（有时叫 **key animator**）。这些有经验的画师并不画出动画的每一格，而是绘制更重要的帧。比如说一个极端的运动或姿势，体现一个场景的本质。如果要表现一个人物投掷一个球的动画，关键帧是手臂最后端时候的帧，手臂在弧线最顶端的帧，和人物释放球体的帧。

然后，**key animator** 会转移到新场景 而 另一个 **in-betweenner**（有时叫 **rough in-betweenner**）会算出关键帧之间的时间间隔，并完成这些关键帧之间帧的绘画。比如一个一秒钟的投掷动画，每秒 12 帧，他们需要指出怎样在首席动画师绘制的关键帧中间完成剩下的 9 帧。

三维关键帧动画的概念也是一样。你有动作中**关键**位置的顶点数据，然后插值算法担当 **rough in-betweenner** 的角色。**插值**将是你三维动画里面用到的最简单的数学算法。

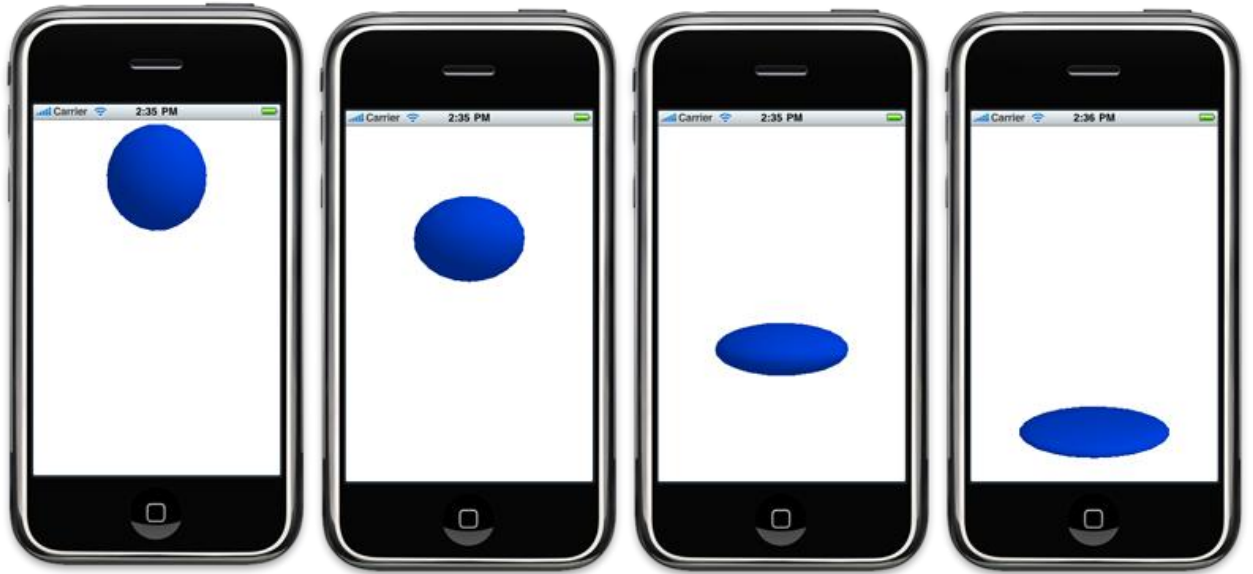
或许我们看一个实际的例子会更明白一点。让我们只关注一个顶点。在第一个关键帧,假设是在原点(0, 0, 0)。第二个关键帧,假设那是在(5, 5, 5),并且在这两个关键帧之间的时间间隔是五秒(为了计算方便)。

动画的一秒钟，我们只需要表现出这一秒前后两个顶点在每个坐标轴上的变化。所以，在我们的例子中，两个关键帧在 x, y, z 轴总共移动了 5 个单位（5 减去 0 等于 5）。一秒钟的动画走了 1/5 的路程，所以我们添加 5 的 1/5 到在第一关键帧的 x, y, z 轴上面，变成(1, 1, 1)。目前数值算出来的过程并不优雅，但是数学算法是一样的。算出总距离，算出与第一关键帧之间流逝的时间比例，两种相乘再加上第一关键帧的坐标值。

这是最简单的插值，叫**线性插值**，适用于大部分情况。更加复杂的算法，要权衡动画的长度。例如在 **Core Animation** 中，提供了几种"ease in", "ease out", or "ease in/out"等几种选项。也许我们会在以后的文章中讨论非线性插值。不过现在，为了保持简单易懂，我们继续讨论线性插值。你可以通过改变关键帧的数量和它们的时间间隔，完成绝大多数动画。

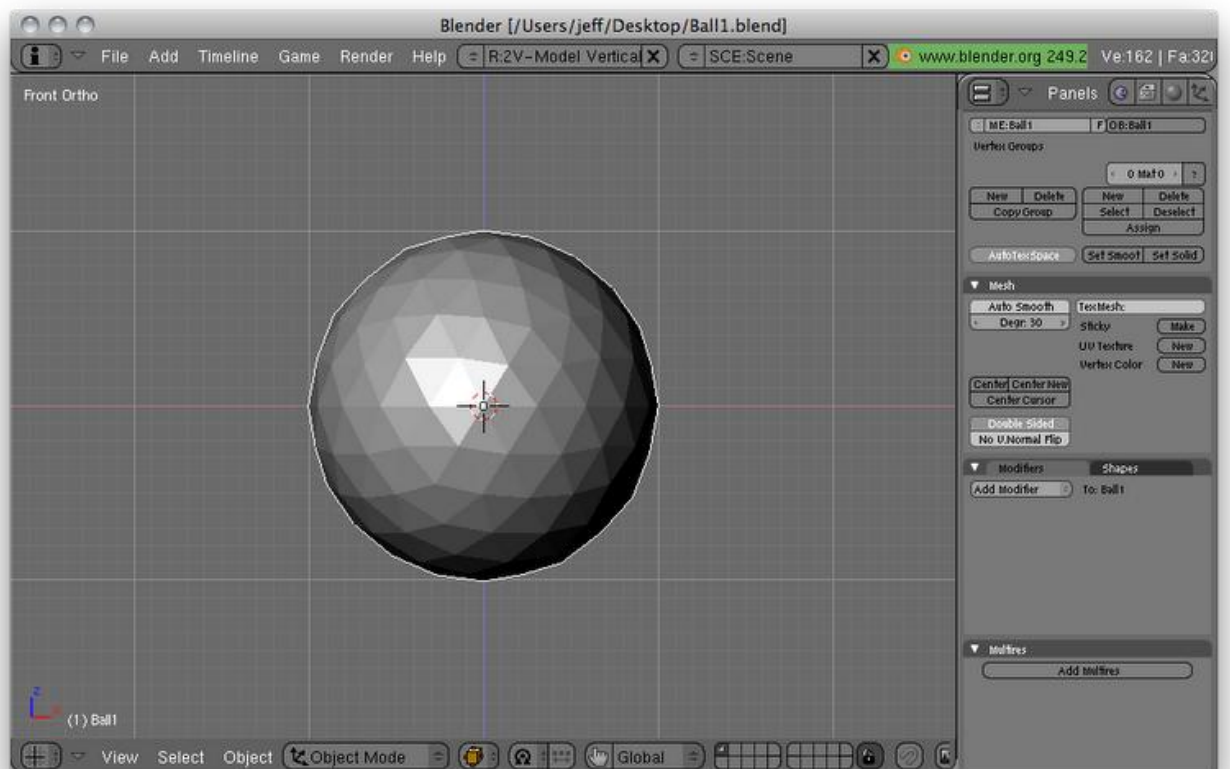
Keyframe Animation in OpenGL

让我们看一个 OpenGL 中简单动画的例子。当一个传统的手工绘画师被训练以后，他们做的第一件事情就是做一个能够被挤压的而且正在反弹的小球。这同样适合我们，程序会像下面这样：

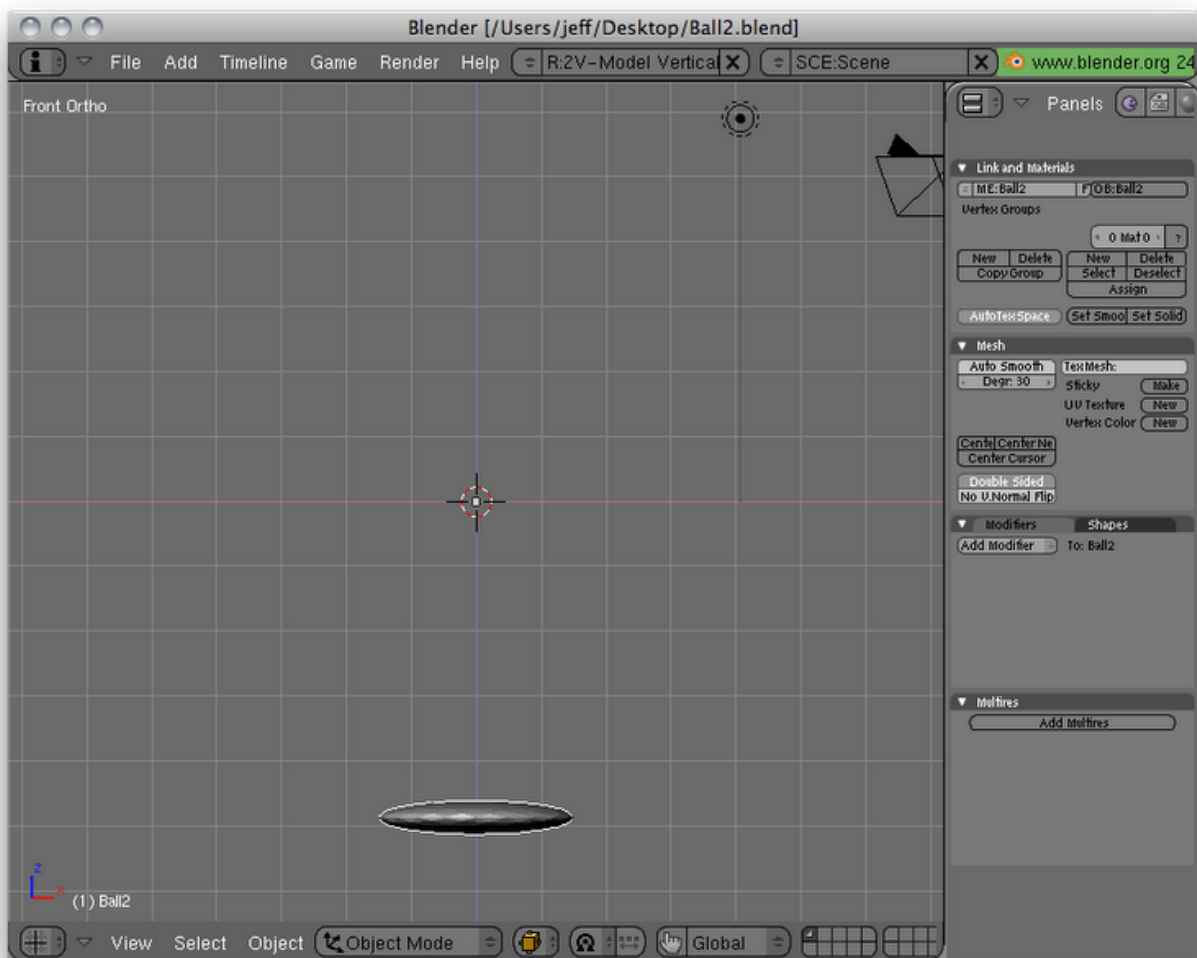


让我们用 [Blender](#)（或者任何你想用的 3d 程序，如果你有方法输出 **vertex**，**normal data** 的数据用人工的方法。在这个例子里面我会用 [Blender export script](#)，它能生成一个有顶点数据的头文件）创建一个球。

我开始在原点创建一多面体，并且重新命名为 **Ball1**，然后我保存这个文件。使用我的脚本渲染并且输出 **ball1**。你可以在[这里](#)找到这个帧的[渲染文件](#)。



现在，我们按另存为（F2）保存一个 **Ball2.blend** 的副本。我重命名为 **Ball2** 以便于输出脚本使用不同的名字命名数据类型。接着点击 **tab** 键进入编辑模式，点击 **A** 移动和缩放球体上的点，直到球体被压扁。保存压扁的球然后输出到 **Ball2.h**。你可以在[这里](#)找到压扁的球的[资料](#)。



到这里，我们有两个头文件，每个文件里面都包含着我的动画里面要用到的每个帧的顶点数据。从 [my OpenGL ES template](#) 开始工作，我先在 `GLViewControler.h` 定义了一些新的值，它能帮助我追踪小球的运动。

十四元数

在进入下一篇关于骨骼动画的文章之前，让我们先花点时间来了解一个马上会使用到的新数据类型：**四元数**[译者注:关于四元数的概念可以参考这个链接:[点我](#)]。我们用四元数存储单一骨骼在 3 个轴线上的旋转信息，换句话说，存储的是骨骼指向的方向。在下一部分介绍的仿真骨骼动画中，你将会看到，模型的顶点是同一个或多个骨骼相关联的，当骨骼移动时它们也会随之变化。相对于将[欧拉角](#)信息存储在 3 个 `GLfloats` 变量或一个 `Vector3D` 变量里来说，使用四元数有 2 个优点：

1. 四元数不会造成万向节死锁([gimbal lock](#))，但是欧拉角容易造成万向节死锁，使用四元数能够让我们的 3D 模型能够全方位的移动。
2. 相比于给每个欧拉角做矩阵旋转转换计算，使用四元数结合多角度旋转可以显著的减少计算量。

从某些方面来看，四元数极其复杂且难于理解。它们是高级数学：完全疯狂的符咒。幸运的是，你不需要完全理解它们

背后的数学含义。但是，我们现在需要使用它们来完成骨骼动画，所以还是值得我们花费些时间来讨论下它们的概念和怎么使用它们。

Discovery 探索

从数学上讲,四元数是复数的一个扩展延伸，于 1843 年由 [Sir William Rowan Hamilton](#) 发现。技术上讲，四元数表现为实数之上的 4 维正规可除代数。**Zoiks!**更简单的讲，四元数被认为是第四维度用来计算笛卡尔坐标中的 3 个坐标值。好吧，一切可能不那么简单，对吧？

先别怕，如果你不精通高等数学，四元数可能会让你头疼。但是，如我之前所说，如果你只是使用它们，完全不必深入了解。这玩意和你见过的一些概念是非常类似的。不知你是否还能想起我们在 3 维空间里涉及到的 4X4 矩阵的矩阵转换。当我们使用已转换的数据的时候，忽略了第 4 个值。我们可以把这里的第四个值当成四元数，为计算提供了一个位置。数学范畴内，请不要跟我说——过度简化有助于凡人在四元数世界里占有一席之地，有所作为。

四元数在探索时代里被认为是相当创新的，但最繁荣的时期却如此短暂。在 1880 中期，[向量微积分](#)开始在计算领域取代四元数理论，因为它用了一种更为容易理解和描述的概念描述了同样的现象。

Not Quite Dead Yet! 虽死犹生

但在 20 世纪，四元数又重新获宠。正如我们在 [part 7](#) 里讨论的，有一个被称为 gimbal lock 的现象，当你在每个轴线单独做旋转转换的时候就会发生，此现象的危害就是可能导致在三个轴中的一个轴上停止旋转。

尽管事实是四元数源于复数和理论数学，但它们都有实际应用。其中一个实际应用是三轴线上旋转角的展现。由于四元数用四个维度展示了笛卡尔（或三轴）旋转，此展现不会导致 gimbal lock,而且你可以在四元数和旋转矩阵之间，四元数和欧拉角之间进行无损转换。这使得存储某些对象的旋转信息相当完美，比如。。。骨骼框架中的单独骨骼？不需要存储 3 轴的角信息，而是存储一个单独的四元数。

四元数和矩阵一样，可以相乘，且存储于不同四元数中的旋转值通过相乘来合并计算。四元数乘积和 2 个旋转矩阵乘积的结果是完全一样的，考虑到减少计算量，这意味着除了要避免 gimbal lock,还要减少每次程序循环运行的 [FLOPS](#)(每秒浮点运算次数)。和矩阵乘法相比，四元数乘法不仅步骤少，而且可以通过一个四元数表达 3 轴所有数据。如果通过 [Vector3D](#) 或 3 个 [GLfloats](#) 来存储旋转信息，我们经常不得不做 3 次矩阵乘法——每轴都要算一次。结论是，通过把存储旋转的独立角信息存为四元数，可以带来可观的性能提升。

附录

setupview 重写

我在[从零开始学习 OpenGL ES 之四 - 光效](#)一文中使用了一个普通 GLfloat 数组。由于它没有使用任何非 OpenGL 定义的数据结构，所以是最为普通和方便的方式。

但在此我使用在[第一部分](#)中定义的 [Vertex3D](#), [Vector3D](#) 和 [Color3D](#) 数据结构重写了 `setupView:`方法。并不是这种方法“更好”，但是它是一种不同的方式。当我第一次学习 OpenGL 时，我发现使用顶点，颜色和三角形的术语比可变长度浮点数组更容易理解。如果你和我一样，那么你会发现这个版本更容易理解。

除了使用自定义数据结构外，我还减少了环境光元素的数量并将光源向右移动了一点。然后使用 `Vector3DMakeWithStartAndEndPoints()` 将移动的光源指向二十面体。这样做使得光效更为生动一点。

```
-(void)setupView:(GLView*)view
{
    const GLfloat zNear = 0.01, zFar = 1000.0, fieldOfView = 45.0;
    GLfloat size;
    glEnable(GL_DEPTH_TEST);
    glMatrixMode(GL_PROJECTION);
    size = zNear * tanf(DEGREES_TO_RADIANS(fieldOfView) / 2.0);
    CGRect rect = view.bounds;
    glFrustumf(-size, size, -size / (rect.size.width / rect.size.height), size /
               (rect.size.width / rect.size.height), zNear, zFar);
    glViewport(0, 0, rect.size.width, rect.size.height);
    glMatrixMode(GL_MODELVIEW);

    // Enable lighting
    glEnable(GL_LIGHTING);

    // Turn the first light on
    glEnable(GL_LIGHT0);

    // Define the ambient component of the first light
    static const Color3D light0Ambient[] = {{0.05, 0.05, 0.05, 1.0}};
    glLightfv(GL_LIGHT0, GL_AMBIENT, (const GLfloat *)light0Ambient);

    // Define the diffuse component of the first light
    static const Color3D light0Diffuse[] = {{0.4, 0.4, 0.4, 1.0}};
    glLightfv(GL_LIGHT0, GL_DIFFUSE, (const GLfloat *)light0Diffuse);

    // Define the specular component and shininess of the first light
    static const Color3D light0Specular[] = {{0.7, 0.7, 0.7, 1.0}};
    glLightfv(GL_LIGHT0, GL_SPECULAR, (const GLfloat *)light0Specular);
    glLightf(GL_LIGHT0, GL_SHININESS, 0.4);

    // Define the position of the first light
    // const GLfloat light0Position[] = {10.0, 10.0, 10.0};
    static const Vertex3D light0Position[] = {{10.0, 10.0, 10.0}};
    glLightfv(GL_LIGHT0, GL_POSITION, (const GLfloat *)light0Position);

    // Calculate light vector so it points at the object
    static const Vertex3D objectPoint[] = {{0.0, 0.0, -3.0}};
    const Vertex3D lightVector = Vector3DMakeWithStartAndEndPoints(light0Position[0], objectPoint[0]);
    glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, (GLfloat *)&lightVector);
```

```
// Define a cutoff angle. This defines a 50° field of vision, since the cutoff
// is number of degrees to each side of an imaginary line drawn from the light's
// position along the vector supplied in GL_SPOT_DIRECTION above
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 25.0);

glLoadIdentity();
glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
}
```

你可以随意调整光的属性，增加额外的光源或二十面体，体验一下这些调整会为场景带来什么样的变化。这些东西是很难体验出来的，所以不要指望一晚上就理解了所有东西。