西蒙 iphone-OpenGL ES 教程

西蒙 iphone-OpenGL ES 教程-01

受到花生大大的鼓励，继续努力翻译给大家看

西蒙 iphone-OpenGL ES 教程

图元是构成复杂物体的基本绘图要素。在 OpenGL ES 中，你可以使用的图元有点，线，三角形。它们都有非常强的自我解释性，我觉得你需要有些例子让你看到它们。

首先，让我们来看看一些代码，然后我们可以谈论这是怎么回事，这样您就可以使用它来创建一些自己的代码。

图元 ＃1 — 三角形
三角形是最“复杂”的图元，但是很容易使用，并且非常实用,这将是你可以绘制的第一个 OpenGL 的图元。当我们绘制一个三角形的时候，我们需要告诉 OpenGL 在3d 空间中的三角形的3系坐标，并且，OpenGL 将非常顺利的渲染这个三角形。

在开始之前，复制00教程中的项目代码或者从这里下载下项目代码： AppleCoder-OpenGLES-00.tar.gz .在 XCode 中打开，开启 EAGLView.m 文件，找到 drawView 函数。这里就是施展魔法的地方。

首先，我们需要定义一个三角形。要做到这点，我们需要知道在我们要处理的坐标的两种类型：模型和世界。模型坐标是指我们正在绘制的实际图元，世界坐标告诉 OpenGL 观察者在哪里。（在世界坐标中，观察者一般在（0.0,0.0,0.0）的地方）

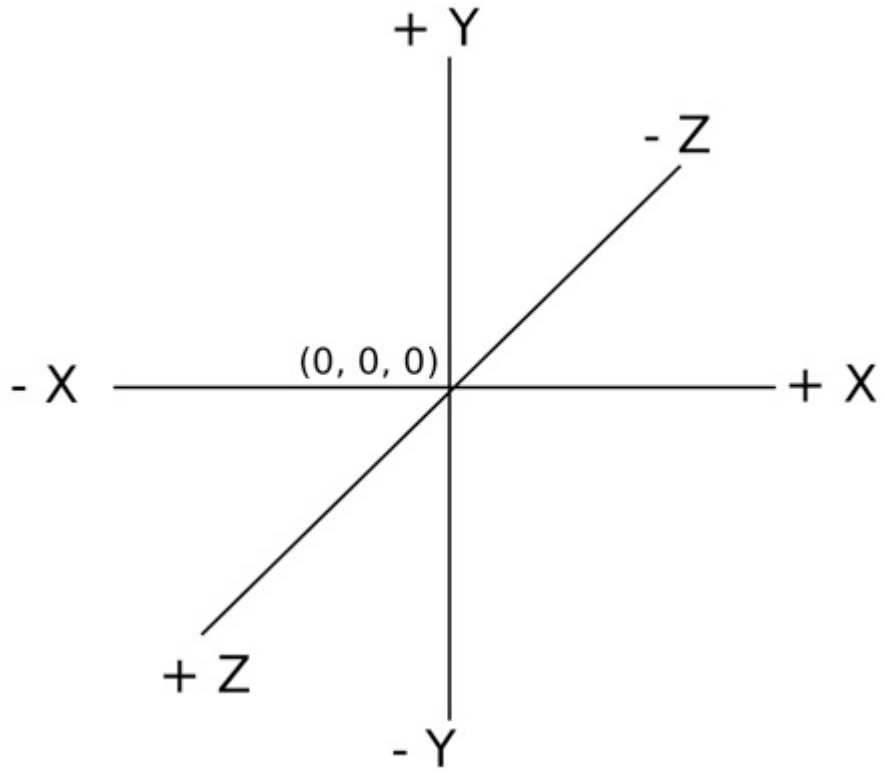
第一个例子将说明这点。首先，我们定义这个三角形在模型空间使用3 x 3d 坐标（x,y,z）：

```
const GLfloat triangleVertices[] = {
        0.0, 1.0, -6.0,// Triangle top centre
        -1.0, -1.0, -6.0,// bottom left
        1.0, -1.0, -6.0,// bottom right
    };
```

如上所示，这里使用了3个坐标来表示一个三角形，需要注意的是，我们定义三角形顶点是逆时针来显示的。虽然描述三角形的可以用逆时针也可以用顺时针，但是我们必须和上述一样用逆时针来描述三角形。不过，我建议你用逆时针来描述三角形，因为我们以后可以用逆三角形来达到一些先进的功能。

（补充：逆三角形在3d 中被认为是正面，而顺三角形则被认为是反面。在纹理渲染中被使用到）

虽然本教程应该是纯粹的 iPhone OpenGL ES 的，对于初学者来说，我会简要的描述三维坐标系统。看看这张图片：



对于我的绘画技巧，我深表遗憾。不过这个图代表了模型空间和世界空间。 试想一下，这是您的计算机屏幕， X 和 Y 的横向和纵向的，你应该预料到， Z 表示深入。这个中心位置就是(0.0,0.0,0.0).

所以，看我们的三角形中所描述的顶点上述情况，第一点（ 0.0 ， 1.0 ， -6.0 ）中心将在的 Y 轴，上涨1点，在屏幕的深度为6点。 第二个坐标是右边的 Y 轴1.0点，低于 X 轴（因此-1.0的 Y 值） ，仍然回到屏幕-6.0点。这同样适用于第三个坐标。

为此，我们确定目标在我们眼睛之前(z 值是负的),所以这个目标是可见的（记得吗，观察者或者说照相机是在（0.0，0.0，0.0）的位置上）所以说 OpenGL 的深度测试是失败的并且它没有被渲染。

我可以听到你尖叫“嘿，我还以为你说，这是模型坐标不是世界坐标！ ”。 是的，这是对的，但是，当我们去渲染这个三角形之前， OpenGL 的将只是把对象放在（ 0.0 ， 0.0 ， 0.0 ） 的位置上。 因此，我们将它放到屏幕内才可见。 当我们进入转换（移动，旋转等），您会看到，您将不必设置对象 Z 值为负，也可以使之可见。在此之前，让目标的 Z 坐标在-6.0的位置上。

绘制函数：

所以我们这样做是为了说明目前的三角形。我们现在需要告诉 OpenGL，数据保存在哪里，以及如何去绘制它。这个过程只需要很少的几行代码。回到 drawView 函数，并且实行以下的代码：

```
- (void)drawView {

const GLfloat triangleVertices[] = {
        0.0, 1.0, -6.0,                    // Triangle top centre
        -1.0, -1.0, -6.0,                  // bottom left
        1.0, -1.0, -6.0                    // bottom right
    };

    [EAGLContext setCurrentContext:context];
    glBindFramebufferOES(GL_FRAMEBUFFER_OES, viewFramebuffer);
    glViewport(0, 0, backingWidth, backingHeight);

// -- BEGIN NEW CODE

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

glVertexPointer(3, GL_FLOAT, 0, triangleVertices);
glEnableClientState(GL_VERTEX_ARRAY);
glDrawArrays(GL_TRIANGLES, 0, 3);

// -- END NEW CODE

    glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
    [context presentRenderbuffer:GL_RENDERBUFFER_OES];

[self checkGLError:NO];
```

}

如你所见，这4行代码就是我们渲染一个三角形的。让我们从上往下一行行的打断这些代码来分析，你会发现它们是非常的简单。

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

这行代码只是简单的清除了屏幕。这个控制位告诉 OpenGL，我们使用上次教程里面 setupView 函数中设置的颜色（黑色）来清除屏幕，并且清除了深度缓冲。 请注意，如果我们没有创建深度缓冲和开启深度缓冲（如我们应该做的）,这个屏幕将不会渲染。如果我们不使用深度缓冲，我们将不需要通过 glClear()里的 GL_DEPTH_BUFFER_BIT.

因此，我们无论是清除以前绘制的这个缓冲区（请记住，这是双重缓冲动画;利用一个缓冲区而另一个缓冲区显示） 。

glVertexPointer(3, GL_FLOAT, 0, triangleVertices);

此 函数告诉 OpenGL 的情况下我们的数据是什么格式，它是有4个参数，这个功能是很简单细分：

1.Size-这个值表示了每个坐标有几个数字。我们现在是3，是因为是3d 坐标(x,y,z).如果我们使用2d 绘制，不加入深度(就是 z),我们只要在这里写2就可以了。

2.Data Type- GL_FLOAT 意味着我们用 float 点值。 您也可以使用整数值，但如果你想你需要习惯于使用浮点值的3D 世界的浮点运算。

3.Stride - 这个 stride 告诉 OpenGL 在每个坐标之间忽略哪几个点。别对这个表示疑惑，保持它是0就行了。你使用它
当您载入顶点数据文件的格式有更多的填充数据或肤色的数据，也就是说，一个3D 程序像搅拌机。

4.Pointer to the Data – 数据本身，正是因为它，三角形才会出现。

因此，我们告诉 OpenGL 清除缓冲区，告诉它的数据是我们的目标和它的格式，现在我们需要告诉 OpenGL 的东西很重要：

glEnableClientState(GL_VERTEX_ARRAY);

OpenGL 是一个状态机。 这意味着你打开和关闭功能的要求就是启用和禁用命令。之前，我们使用过 glEnable()，这影响到 OpenGL 的服务。 glEnableClientState ()影响到我们的程序方面（就是客户端面）。所以我们要做的就是告诉 OpenGL 我们顶点数据是一个顶点数组并且转换到 OpenGL 的绘制顶点功能。在这个情况下，顶点可以是一个颜色数组，我们将呼叫 glEnableClientState (GL_COLOR_ARRAY)或者一个纹理坐标数组如同纹理映射。(别垂涎三尺，你需要掌握所有的基本知识包括纹理映射)

随着我们进一步的研究 OpenGL ，我们会使用不同的客户端状态，这将使用变得更为清晰。

现在命令 OpenGL 渲染一个三角形。

glDrawArrays(GL_TRIANGLES, 0, 3);

一旦这个方法被调用，OpenGL 将用我们之前的两个函数的信息开始执行。屏幕中间出现一个白色的实心的三角形（白色是默认的绘制颜色）。现在的三角形是一个实心的，如果你需要一个镂空的三角形，你将要用不同的绘制方法。

分析这个方法里的三个参数：

1.Drawing Method- 在这种情况下，我们已通过 GL_TRIANGLES 这似乎相当明显，因为我们画一个三角形。然而，这一方法的第一个参数将变得很明显，当我们使用此功能，来绘制一个平面体。

2.First Vertex- 我们阵列只有3点，所以我们想要的 OpenGL 提请第一坐标的数组，这里指定

零就像进入一个标准数组。 如果我们有多种原始的顶点数组，我们可以在这里处理。我会在以后的教程教你，当我告诉您该如何建立复杂的物体的时候。现在，这里只要使用0。

3.Vertex Count-这将告诉 OpenGL 在我们的数组中有多少顶点需要被绘制。比如说，我们绘制一个三角形，所以至少要3点，一个平方形需要4点，一个线需要2点（或更多），一个点需要一个（或者多点）

当你把这些代码都输入到你的 drawView 函数里面以后。点击“Build and Go”运行这个程序在模拟器里面。你的模拟器应该看起来象如下的图：

如我们所说的，有一个实心的白色的三角形出现在屏幕的中间。

在我们做其他的图元之前，尝试修改 z 值，你就会明白我所说的是什么意思。如果你把 z 改为0，你将什么也看不到。

如果你自己输入几行代码，我希望你可以来发现 OpenGL ES 是如何工作的。如果你学习过“标准”的 OpenGL 的教程，我希望你可以发现 OpenGL 与 OpenGL ES 的不同。

期待。。。
下个教程将着重于扩大代码量并生产一个平方形。

西蒙 iphone-OpenGL ES 教程-02

OpenGL ES 02 – 绘制图元 2 — 矩形

严格意义上来说，矩形并不是 OpenGL ES 的图元。 但是，让我们面对现实，绘制这些矩形象绘制一个三角形一样非常简单。在本教程中，我们将把基本的三角形绘制代码转化为绘制矩形。再一次说明，这些渲染是静止的，但是我们通过转化（即移动它们）来使它们快速移动。当然，一旦我们完成了矩形，我们将试图完成一个立方体，我们将完成一个纹理映射的立方体。

本教程的快速小结及其他教程

上个教程，我们用空白画布的 XCode 工程来渲染了一个实心的白色的三角形。为了做到这点，你创建了一个顶点数组，告诉 OpenGL 这些数据并且使用 glVertexPointer() 来格式化。为渲染这个顶点数组定义一个状态，并且使用 glDrawArrays ()来渲染它。

今天，我们要使用之前的代码，用制作三角形的方法来制作一个矩形。要做到这点，我们仅仅需要修改一对（组）代码。第一点是显然的，我们需要4点来组成矩形，而不是原来3点的三角形。然后，我们要告诉 OpenGL 用不一样的 glDrawArrays().方法来绘制一个不同的物体。

让我们开始吧。

定义矩形的顶点

打开之前教程中建立的 XCode 工程，找到 drawView 函数。注释掉 triangleVertices ，但是不是删除它，我们在之后转换的时候还需要用到，并且添加下列代码：

```
const GLfloat squareVertices[] = {
        -1.0, 1.0, -6.0,            // Top left
        -1.0, -1.0, -6.0,           // Bottom left
        1.0, -1.0, -6.0,            // Bottom right
        1.0, 1.0, -6.0              // Top right
    };
```

这个定义了我们的矩形。注意，这个矩形的顶点也是逆时针的。然后，到下面的绘制三角形的代码处，将这些代码都注释掉，回复到开始状态。 所以注释掉三个函数调用的 glVertex Array （ ） ， glEnableClientState （ ） ，和 glDrawArrays （ ） ，并添加以下代码：

```
    glVertexPointer(3, GL_FLOAT, 0, squareVertices);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
```

这三项职能相同，只是功能略有不同。

```
    glVertexPointer(3, GL_FLOAT, 0, squareVertices);
```

这个唯一的变化就是我们告诉 OpenGL 使用了不同的顶点设置方案，而不是现在的三角形。

glEnableClientState()同样是告诉 OpenGL 从顶点数组来绘制（而不是颜色数组或者其他的什么）

glDrawArrays(GL_TRIANGLE_FAN, 0, 4);

这里有些改变了。在上个教程，我们使用 GL_TRIANGLES 做用第一个参数，并且使用3在第3个参数里。第2个参数。之前是0，这里也还是0，因为只包含了矩形的顶点。

第一个参数是绘图模式，而且你现在看到两种可能的 OpenGL 绘图方式。 我想花时间来讨论现在不同的绘图模式。

它们是：

GL_POINTS
GL_LINES
GL_LINE_LOOP
GL_LINE_STRIP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN

我们还没有讨论点或线，所以我只介绍最后的三个 。在我开始之前，我要提醒你，顶点数组可能包含不止一个三角形，以便当您只看到一个物体顶点数组，你要知道，不仅限于这一点。

GL_TRIANGLES - 这个参数意味着 OpenGL 使用三个顶点来组成图形。所以，在开始的三个顶点，将用顶点1，顶点2，顶点3来组成一个三角形。完成后，在用下一组的三个顶点来组成三角形，直到数组结束。

GL_TRIANGLE_STRIP - OpenGL 的使用将最开始的两个顶点出发，然后遍历每个顶点，这些顶点将使用前2个顶点一起组成一个三角形。所以 squareVertices[6~8]将与 squareVertices[0~2] 和 squareVerticies[3~5].生成一个三角形。 squareVertices[9~11]将与 squareVertices

[3~5] 和 squareVertices[6~8]生成三角形。

也就是说，0，1，2这三个点组成一个三角形，1，2，3这三个点也组成一个三角形。

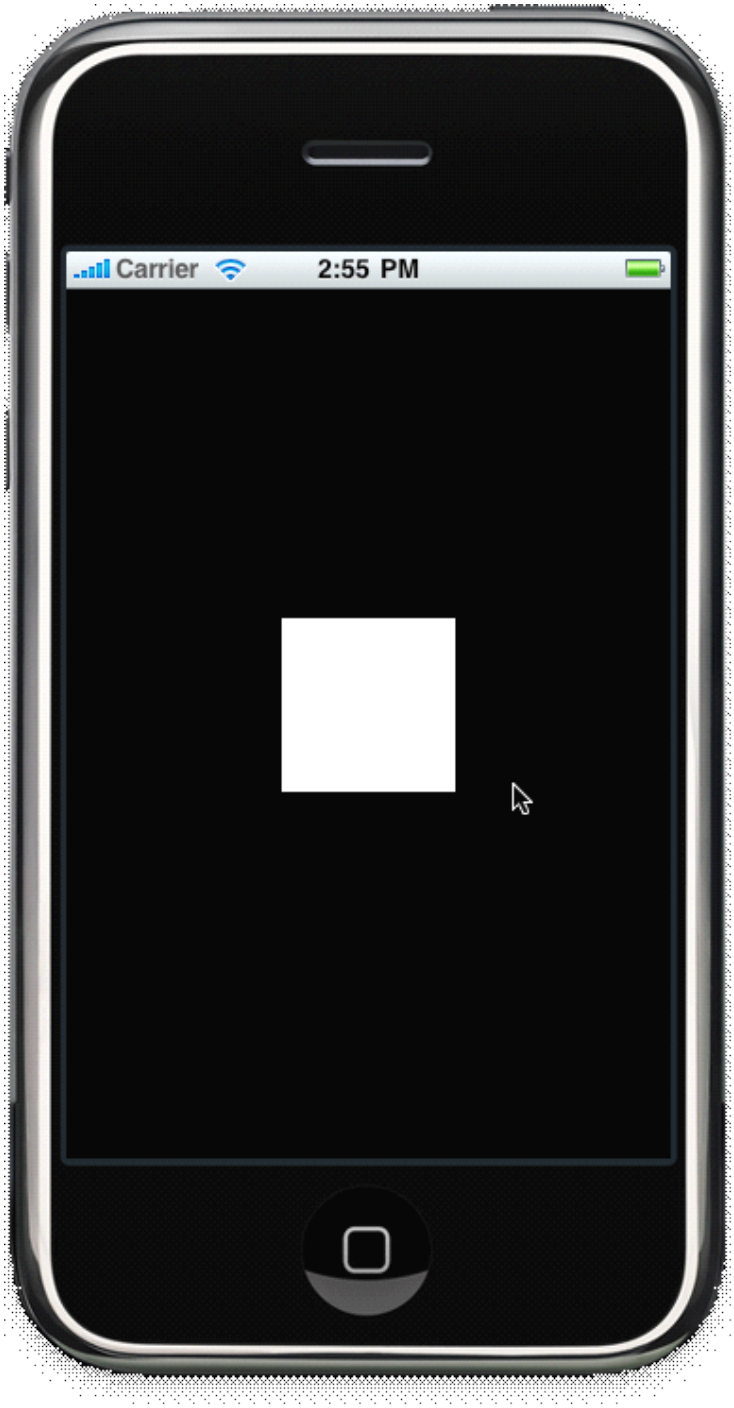
注意的是， squareVerticies[0~2]表示的意思是：
squareVerticies[0] x 坐标
squareVerticies[1] y 坐标
squareVerticies[2] z 坐标

如果我没有说清楚的话，我用下面的例子来说明。

GL_TRIANGLE_FAN - 在跳过开始的2个顶点，然后遍历每个顶点，让 OpenGL 将这些顶点于它们前一个，以及数组的第一个顶点一起组成一个三角形。 squareVertices[6~8]将与 squareVerticies[3~5] （前一个）和 squareVerticies[0~2]（第一个）.生成一个三角形。

也就是说，同样是0，1，2，3这4个顶点。
在 STRIP 状态下是，0，1，2；1，2，3这2个三角形。
在 FAN 状态下是，2，1，0；3，2，0这2个三角形。

这次我们将使用 GL_TRIANGLE_FAN ，我们将在显示区域获得一个矩形。点击”Build & Go”,然后你将在屏幕上看到一个白色的矩形（平面）



回头看你的顶点数组。设法想像用三角形来绘制一个矩形。OpenGL 就是这样做的。

三角形点1： squareVerticies[0~2] －矩形的左上

三角形点2： squareVerticies[3~5] －矩形的左下

三角形点3： squareVerticies[6~8] －矩形的右下

考虑到上述3点，一个三角形的 OpenGL 的绘制将弥补左下角的一半面积。试想矩形被对角线分割为左上角和右下角。如何通知这两个三角形？OpenGL 刚才只绘制了矩形的左半边。

三角形点1： squareVerticies[9~11] －矩形的右上
三角形点2： squareVerticies[6~8] －矩形的右下，上个点
三角形点3： squareVerticies[0~2] －矩形的左上，第一点

只使用了一个新的点，OpenGL 可以渲染这个三角形来完成矩形。

GL_TRIANGLE_STRIP

返回代码，改变 glDrawArrays() 的第一个参数由 GL_TRIANGLE_FAN 为 GL_TRIANGLE_STRIP:。

点击”Build & Go”然后你会获得下面的图片。

让我们来看看，为什么我们只是单单修改了下绘制模式，我们却没有获得一个矩形。 OpenGL 展示我们的顶点数组如下：

三角形点1： squareVerticies[0~2] －矩形的左上
三角形点2： squareVerticies[3~5] －矩形的左下

三角形点3： squareVerticies[6~8] 一矩形的右下

OpenGL 使用了前3个点来渲染一个三角形。这个左下的三角形和之前是相同的。

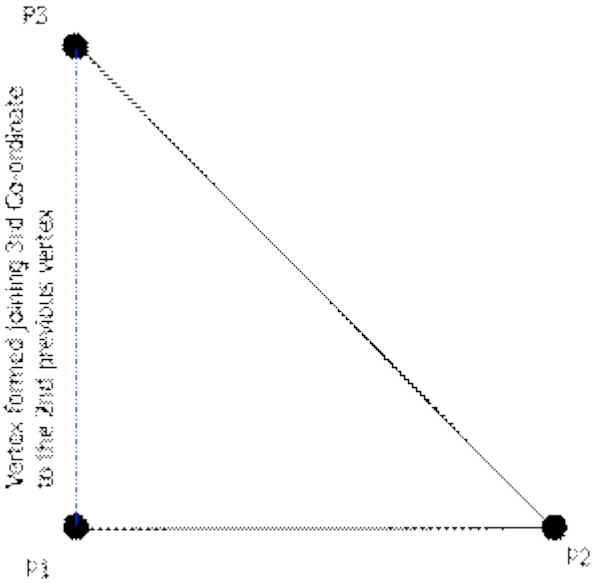
三角形点1： squareVerticies[9~11] 一矩形的右上
三角形点2： squareVerticies[6~8] 一矩形的右下，上个点
三角形点3： squareVerticies[3~5] 一矩形的左下，上2个点

这个是现在渲染三角形的3个点。这个三角形和我们理想的三角形有90度的夹角。

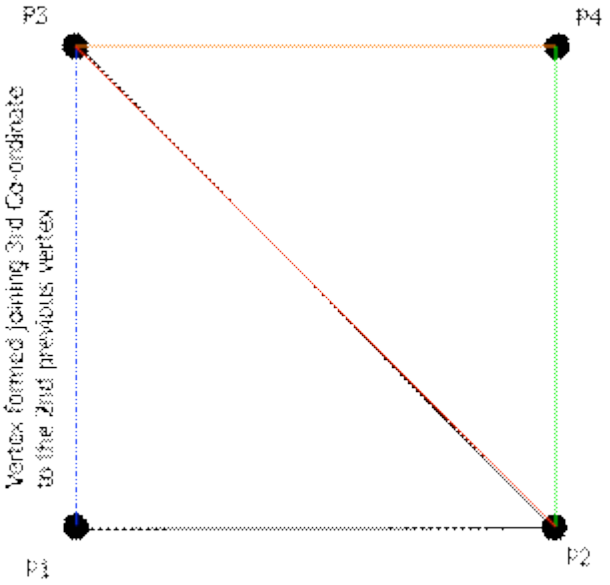
如果我们提供的顶点的数组不同，我们可以实现一个正确的 GL_TRIANGLE_STRIP ，我们现在还是做我们的 GL_TRIANGLE_FAN 。必须记住，你的绘制模式必须和你的顶点数组保持一致，否则，就会出现奇怪的效果，就象我们刚才修改的那样。
如果，你坚持使用 GL_TRIANGLE_STRIP ，你只需要修改你的顶点数组，如下：

```
const GLfloat stripSquare[] = {
        -1.0, -1.0, -6.0,          // bottom left
        1.0, -1.0, -6.0,           // bottom right
        -1.0, 1.0, -6.0,           // top left
        1.0, 1.0, -6.0             // top right
    };
```

因此，与上述情况，我们可以看到第一个三角形将形成头三个顶点，产生一个三角形如下：



现在，通过指定点右上角顶点（ P4） ，一个新的三角地带将形成与左上角工作（ P3 ）和第二前顶点（P2）这是右下角。新的顶点显示为橙色，绿色和红色如下：



结果是，那种方式都可以让我们生成一个矩形。最终的结果是相同的。但这些提醒我们，要注意保持你的顶点数组和你的绘制模式的一致。

最后。。。

现在，你已经知道如何生成三角形和矩形。我的介绍没有包含点和线，因为它们比较简单。在下一教程，我们将增加一些颜色。

一旦我们可以为我们的目标上色，我们可以移动他们，并且在3d 中实现纹理映射。虽然，它不会象 Doom 3一样酷。但你知道，你可以开始建立3d 的物体，我将开始包含3d 世界的知识。

西蒙 iphone-OpenGL ES 教程-03

OpenGL ES 03 – 转化

今天，我们要在之前的基础上，在屏幕上同时显示三角形和矩形。为了做到这点，我们需要移动它们。移动物体这个动作我们称之为转化。（坐标转换）

在 OpenGL ES 中，对模型 / 物体进行转换有三种不同的类型。它们是：

1.Translate – 在3d 空间里移动物体。
2.Rotate – 在 x,y,z 坐标轴上进行旋转。
3.Scale – 改变物体的尺寸。主要应用于在3d 空间中的2d 投影体。当然，也可以应用于特殊效果。

为了展示这些不同的方法，我们使用 translate 函数在屏幕上同时生成三角形和矩形，以后在着手做其他2个（方法）。

Translate
为了显示移动的效果，OpenGL ES 给了我们一个单一的方法供我们使用，这个方法叫： glTranslatef()。注意到函数的结尾是‘f’了吗？这意味着，我们要提供给 OpenGL 一个浮点数。OpenGL ES 也提供了使用16进制的方法，方法叫： glTranslatex()。16进制的写法应用于那些没有浮点运算的硬件，幸运的是 iPhone 本身就有浮点运算，所以我们不需要使用16进制的算法，直接用浮点运算就可以了。

我觉得需要指出的是，如果你在 XCode 里面用16进制的算法完成你的工程，你是不会发现有什么不同的（与浮点运算比较）

ok，让我们开始删除一些代码。开启 XCode，打开你的工程。我希望你昨天只是屏蔽了三角形数据和它的渲染而不是删除了它们，否则的话，你只能重新再写一遍了。
首先，让我们来看看2个顶点数组。我们对这些顶点的 z 轴做一些改变，把 z 轴改为0，就象如下所示：

```
const GLfloat triangleVertices[] = {
        0.0, 1.0, 0.0,                  // Triangle top centre
        -1.0, -1.0, 0.0,                // bottom left
        1.0, -1.0, 0.0                  // bottom right
```

```
    };

const GLfloat squareVertices[] = {
        -1.0, 1.0, 0.0,                 // Top left
        -1.0, -1.0, 0.0,                // Bottom left
        1.0, -1.0, 0.0,                 // Bottom right
        1.0, 1.0, 0.0                   // Top right
    };
```

你想起来我们之前为什么要把 z 设置为－6了吗？这示因为我们要把物体放置再屏幕的里面，因为我们的照相机是在(0.0,0.0,0.0)的位置上。现在我们将使用 glTranslatef() 函数来将物体后退6点，而不是象之前那样在顶点中修改。

首先，我们将告诉 OpenGL 我们将使用移动： 无论是投影（世界观）或对象（模型内的世界） 。在这样的情况下，我们要告诉 OpenGL 来生成一个三角形和一个矩形。在 drawView 函数中呼出 glClear()函数。

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);
```

函数 glMatrixMode 告诉 OpenGL,我们工作在顶点模式下，而不是投影。 在我们 setupView 方法，我们使用同样的函数，但使用 GL_PROJECTION 枚举作为参数。OpenGL 是一个引擎，它会保持在一个状态，除非你告诉它要进行改变。所以，矩阵模型将维持在 GL_PROJECTION 状态，直到我们说到模型视窗里面调用 glMatrixMode(GL_MODELVIEW)。现在我们将保持 OpenGL 的状态为 GL_MODELVIEW ，直到我们告诉它改变。

实际上，如果我们希望达到最佳效果，我们应该在我们第一个教程中的 setupView 函数调用之后使用它。另外，我们是在教程中，而不是现实做项目，所以，此刻我们就在 drawView 函数中使用它就行了。

我知道我还没有真正的涉及 OpenGL 中的投影，所以，你如果没有完全理解的话，不要紧张。我们要做的就是在屏幕中放上一些物体，让你一边玩弄这些物品一边学习 OpenGL ES.

现在，去掉绘制三角形代码的屏蔽，象下面一样：

```
glVertexPointer(3, GL_FLOAT, 0, triangleVertices);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_TRIANGLES, 0, 3);
```

在这三行代码之前，再增加两行：

```
glLoadIdentity();
glTranslatef(-1.5, 0.0, -6.0);
glVertexPointer(3, GL_FLOAT, 0, triangleVertices);
        glEnableClientState(GL_VERTEX_ARRAY);
        glDrawArrays(GL_TRIANGLES, 0, 3);
```

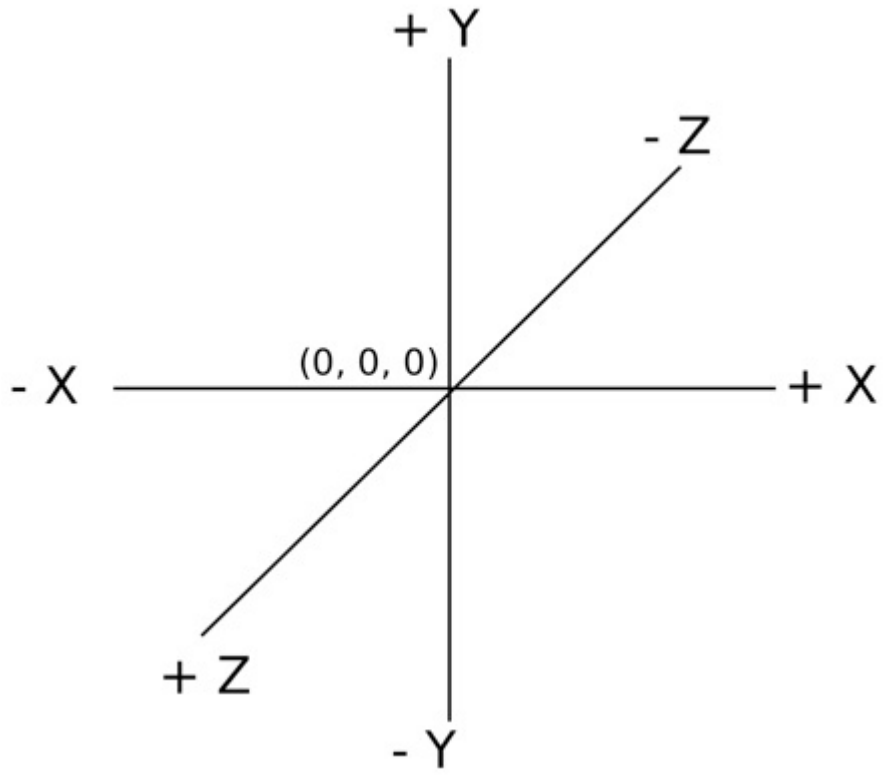
glLoadIdentity() 只是一个覆盖（重置）函数，它只是基本重置所有的条件（重置所有状态）。所以如果我们不调用这个功能的话， glTranslatef() 函数将会移动物体不断的向左和里面，直到物体消失。实际上，我可以使用另外一种更好的函数来达到这点（下一个教程），但是现在我们只需要重置目标数据，所以它就足够了。

下个函数的调用使这里发生变化。

glTranslatef() 函数有3个参数：

glTranslatef(GLfloat xtrans, GLfloat ytrans, GLfloat Ztrans);

在继续之前，在看下我画的这个3d 空间。



记住，照相机在(0.0,0.0,0.0)的位置。我们呼叫 glTranslatef() 使用以上的三个函数。

xtrans = -1.5

ytrans = 0.0

ztrans = -6.0

下面，我们将重新调用我们绘制三角形和矩形的函数，记住，它们应该在屏幕的中心位置。如果我们按以前的教程来绘制它们，它们可能会点对点的绘制到一起。

因此，为了解决这个问题，我按 x 坐标向左移动1.5个点。看下上面的3维坐标空间，你会发现，x 的左边是负的。所以，-1.5

z 转换为-6，替换我们在顶点数组里的-6

移动矩形

绘制矩形的代码几乎和三角形一样，如下：

```
glLoadIdentity();
    glTranslatef(1.5, 0.0, -6.0);
    glVertexPointer(3, GL_FLOAT, 0, squareVertices);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
```

这次，你通过 glTranslatef()函数里的 xtrans 的值将矩形移动到右侧。

点击”Build and Go”你会在你的屏幕上看到这样的图片：

请注意，y 轴一直在屏幕的中心，所以2个图元并排着。

在我们移动前

尝试修改 glTranslatef 函数里的那3个参数，看看哪些地方改变了。 我花了许多小时只是改变参数来看会发生什么事情。您甚至可以屏蔽掉了 glLoadIdentity （ ） ，并看看会发生什么。

旋转

在结束你的试验以后，把你的代码修改为原始的模式，让我们来快速的看下旋转是怎么回事。我们旋转的模式是2d 的，因为我们的物体是2d 的（尽管它在3d 空间里）。之后，我们将创建一个全3d 的物体，并且在全3d 的空间里去旋转它（并且，我们可以做它的纹理映射）

旋转是很简单的：

glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);

这个方法的时候是很简单的。首先，第一个参数设定了你需要旋转多少度，然后，我们指定轴来进行旋转。

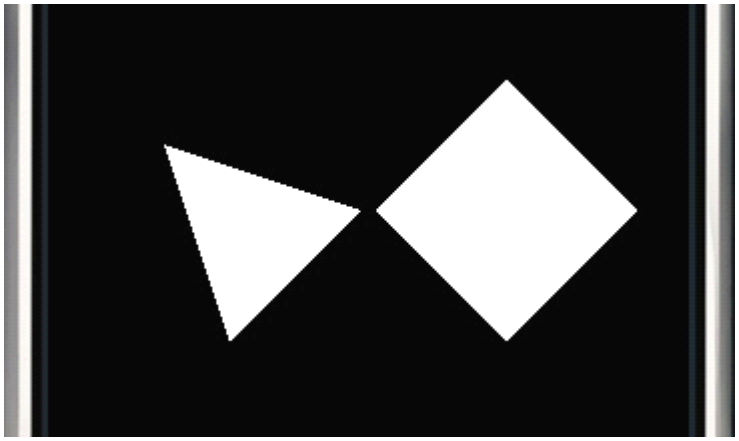
下面，我将师范2种不同的旋转方式。首先，是一个静态的旋转，然后我们将动态的不停的旋转。旋转的目标就是这个三角形和矩形。

首先，让我们做一个简单的旋转，回到 drawView 函数里，对三角形和矩形的绘制函数修改如下：

```
glLoadIdentity();
    glTranslatef(-1.5, 0.0, -6.0);
    glRotatef(45.0, 0.0, 0.0, 1.0);          // Add this line
    glVertexPointer(3, GL_FLOAT, 0, triangleVertices);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_TRIANGLES, 0, 3);

    glLoadIdentity();
    glTranslatef(1.5, 0.0, -6.0);
    glRotatef(45.0, 0.0, 0.0, 1.0);          // Add this line
    glVertexPointer(3, GL_FLOAT, 0, squareVertices);
    glEnableClientState(GL_VERTEX_ARRAY);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
```

我们要做的是，让三角形和矩形沿着 z 轴进行旋转。看到 z 轴那个1.0的参数了没？这就告诉 OpenGL，沿着 z 轴，旋转45度。

看下生成的图像，如下：



就象你开车的时候，轮胎旋转一样，物体沿 z 轴进行旋转。注意了，z 轴是深入屏幕的一条线，所以是沿这条线旋转的。

X 轴旋转的，就如同看着一个旋转的汽车轮胎的赛车走向你（即您在汽车的前面） 。Y 轴旋转的是，轮胎的外观作为司机轮流方向盘，以避免您（希望！ ） 。(就是从上往下看一个旋转的轮胎)。如果你没搞清楚，也别紧张，在下列的师范里，你就会知道，到底发生了什么。

旋转我们的物体

为了旋转我们的三角形和矩形，你需要在每次绘制的时候都增加旋转的角度。到 EAGLView.h 文件，增加一个参数：

```
GLfloat  rota;
```

然后回到 EAGLView.mm 里的 initWithCoder 函数里，增加下面这行。

```
rota  =  0.0;
```

我们做的就是当前旋转角度的变量。
现在，回到 drawView,然后增加下面这行，在 glLoadIdentity() 方法之前。

```
rota += 0.5;
```

我们要做的就是，每次绘制的时候，都增加旋转角度0.5度。修改 glRotatef()函数如下：

```
glRotatef(rota, 0.0, 0.0, 1.0);
```

现在，我们完成了自我增加旋转角度的实现。第一次绘制，将旋转0.5度，第二次绘制，将旋转1.0度。

编译程序，看看轮胎旋转吧。

你自己的实验
在离开本教程之前，我希望你自己做一些实验，比如如下的：

1.改变旋转轴。把 z 轴改为0，将 x,y 改为1。看看沿着其他轴旋转是怎么一回事。

2.将当前的旋转值1.0改为-1.0，注意下，现在它们的旋转和之前的相反了。

3.把当前正的旋转值修改为负的。看看会发生些什么。

我希望你能从中获得些什么。

西蒙 iphone-OpenGL ES 教程-04

OpenGL ES 04 – 颜色及纹理

向我上个教程说的那样，我厌倦了在屏幕上绘制白色的物体，让我们增加一些颜色，就象苹果模块自动生成的工程那样。下面我的介绍你要注意了，因为这些概念将在我们开始纹理渲

染的时候起倒作用（很快的）

在 OpenGL ES 中，可以为整体物体设计一个单一的颜色块， 或可用多色和渲染的颜色转移，以便通过频谱从一个颜色过渡到下一个。在我们的物体上渲染单一的颜色不是很复杂。

象 OpenGL 的所有事情一样，在 OpenGL 中改变颜色是一个“状态”，那就是说，（改变状态）之后，绘制操作将使用这个颜色。 即使我们调用我们的“reset”正在 glLoadIdentity （ ）（这是因为 glLoadIdentity （ ）上只运行实际顶点） 。所以我们只需要增加一行代码，就可以将我们两个物体修改为任意的颜色；没有什么颜色比白色更糟糕了，现在我就来修改为蓝色的。

开启 XCode 并来到 drawView，在第一个 glLoadIdentity（）之后增加下面的：

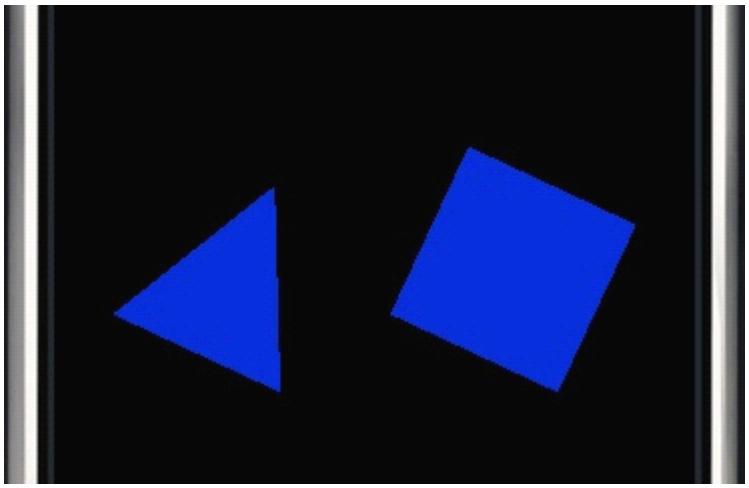
```
glLoadIdentity();
glColor4f(0.0, 0.0, 0.8, 1.0);
```

glColor4f()告诉 OpenGL 开始使用蓝色这个颜色来绘制（填充）。参数如下：

```
glColor4f(   GLfloat red,
             GLfloat green,
             GLfloat blue,
             GLfloat alpha);
```

在 OpenGL ES 中，你必须使用四个参数来定义颜色（RGBA），这里不能使用 RGB 的颜色。
这点你千万别忘了，alpha 是透明度的值，1.0表示实体，0.0表示全透明。
那个 red,green,blue 这三个参数是颜色的浮点值，0.0表示没有强度，1.0表示全强度。

点击编译并运行，你可以看到下面的图片：



比白色的好的太多了，下面让我们看看苹果模板里那彩色的旋转物体是如何实现的。

多重颜色

将一个物体变为多重颜色不需要很多的工作。我们需要定义一个象我们使用过的顶点数组那样的数组，并且告诉 OpenGL 从这个数组中获得颜色。在数组中的每个颜色，都对应了顶点数组里的一个顶点。

让我更清楚的认识到用颜色去渲染矩形。看下面的代码，在这里，我定义了一个颜色数组对应着矩形数组。

```
const GLfloat squareVertices[] = {
        -1.0, 1.0, 0.0,                 // Top left
        -1.0, -1.0, 0.0,                // Bottom left
        1.0, -1.0, 0.0,                 // Bottom right
        1.0, 1.0, 0.0                    // Top right
    };

    const GLfloat squareColours[] = {
        1.0, 0.0, 0.0, 1.0,// Red - top left - colour for squareVertices[0]
        0.0, 1.0, 0.0, 1.0,    // Green - bottom left - squareVertices[1]
        0.0, 0.0, 1.0, 1.0,    // Blue - bottom right - squareVerticies[2]
```

```
        0.5, 0.5, 0.5, 1.0      // Grey - top right- squareVerticies[3]
    };
```

我希望这表示了颜色数组的每个值都对应了矩形顶点中的一个值。在我们通过运行前，我们需要增加一些代码，为了矩形渲染。

```
    glLoadIdentity();
    glTranslatef(1.5, 0.0, -6.0);
    glRotatef(rota, 0.0, 0.0, -1.0);
    glVertexPointer(3, GL_FLOAT, 0, squareVertices);
    glEnableClientState(GL_VERTEX_ARRAY);
    glColorPointer(4, GL_FLOAT, 0, squareColours);        // NEW
    glEnableClientState(GL_COLOR_ARRAY);                  // NEW
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    glDisableClientState(GL_COLOR_ARRAY);                 // NEW
```

这里有三行新的代码，让我们一行行的解释：

glColorPointer(4, GL_FLOAT, 0, squareColours);

这和我们建立矩形顶点数组是类似的，这四个参数是：

1.Size-数组中颜色的数目

2.Data Format-我们在这里使用 GL_FLOAT，因为我们在顶点数组中使用了浮点数。你也可以使用0-255的整形来定义。

3.Stride-再次，这里告诉 OpenGL 在两个值之间跳多少个数字。

4.Arry Points-这里是数据存储的地方。

这里需要注意下数据格式，GL_FLOAT 是告诉 OpenGL 是什么参数格式（是一个枚举类型），GLfloat 是宣布在 OpenGL 中使用什么数据类型。

Ok,这个函数告诉 OpenGL,数据在哪里，格式是什么。但是，就象坐标顶点数组需要告诉 OpenGL 对物体使用坐标，我们需要给 OpenGL 一个必要的状态，告诉 OpenGL 当渲染物体的使用要使用我们的颜色

这就是：

glEnableClientState(GL_COLOR_ARRAY);

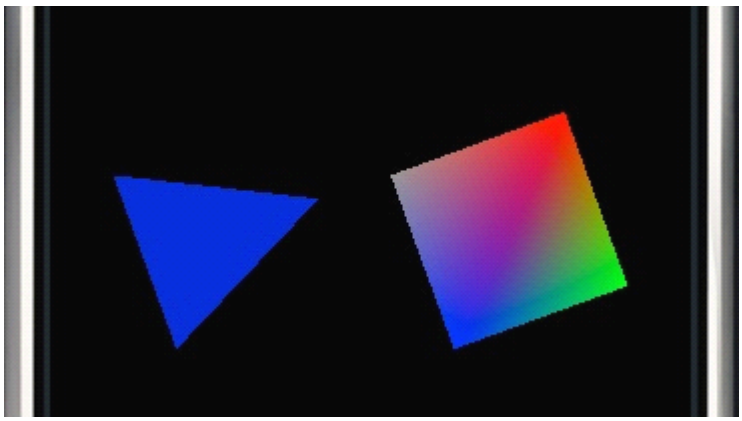
这就能将这个状态添加到 OpenGL 引擎中。不是通过 GL_VERTEX_ARRAY ，我们只需要告诉 OpenGL 颜色数组用 GL_COLOR_ARRAY 。

下一步，我们正常的绘制矩形。在绘制矩形后，我们需要关闭颜色数组，如果我们不这样做，那下个时刻我们绘制三角形的时候，我们还会象矩形那样使用颜色渲染，所以我们调用：

glDisableClientState(GL_COLOR_ARRAY);

这需要关闭 OpenGL 当前的颜色数组状态。如果我们没有这样做，那么第一次调用 drawView 的时候将绘制一个蓝色的三角形，而第二次调用 drawView 的时候将用颜色数组来绘制三角形。 但是，目前只有3个顶点的三角形的统筹阵列（ triangleVertices [ ] ） ，它只会使用前三个颜色。

修改好，运行如下：



如果你喜欢，关闭掉之前的旋转函数，看清楚每个顶点对应的颜色数组。

着色

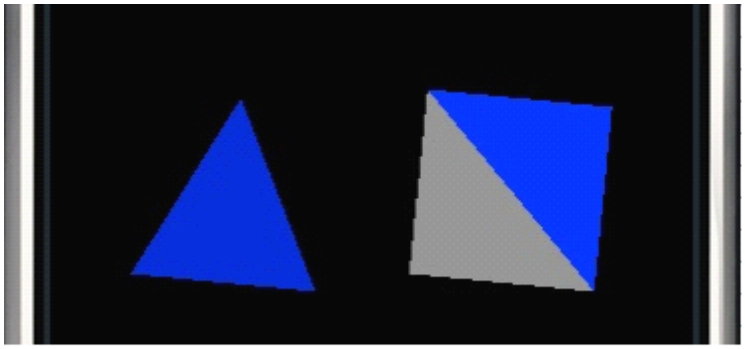
如何通知矩形绘制从一个颜色到下一个？OpenGL 使用了着色。这里有两个着色模型可以在 OpenGL 使用：GL_FLAT&GL_SMOOTH。GL_SMOOTH 是默认的渲染。

为了显示它们的不同，在矩形的 glLoadIdentity （）函数之前，插入下面这行：

glShadeModel(GL_FLAT);

这个 glShadeModel （）函数改变了 OpenGL 中来自平滑的着色模式里的平面着色模式的状态。同样的， OpenGL 的改变它的状态并保持这种状态，直到您告诉它需要修改，所以你可以把这行代码放在 setupView 函数里的任意地方，只要你喜欢。

点击“Build and Go”，通过着色的矩形改变为下面样子：



让我来解释下这里发生了什么。

三角形的渲染是正常的。做为一个平面颜色，着色并不会影响三角形的绘制。在来看看矩形，你现在可以清楚的看到 OpenGL 是用两个三角形来组成一个矩形的。 由于平面着色模式，只使用 OpenGL 的最后肤色填补每一个三角形，那就是 squareColours [2] （蓝色）和 squareColours [3] （灰色） 。如果你不能确定组成矩形的两个三角形的最后顶点是那个，你可以查阅原来的教程。

做一个总结：GL_SMOOTH 是一个平滑着色，这就意味着当你开始填充一个矩形的时候，OpenGL 将使用在我们 squareColours[] 数组里的默认颜色去定义在 squareVertices[] 数组里

的每个顶点。 使用插值中每个像素的面积之间的点，顺利改变颜色之间的四点。换句话说，这将是我们出现彩色的矩形。

GL_FLAT 是使用物体的的最后一个顶点的颜色去填充整个图元。矩形是由两个三角形组成的，所以我们看到2块颜色的三角形。

结论

好吧，我希望这对你是有益的。在现实生活中，你可能只是想使用 GL_SMOOTH 作为着色，除非你做其中的一个怀旧的3D 游戏从 C64 days。 GL_SMOOTH 是预设的，因此您不必启用它。

另外，请注意，你上面使用的颜色分类，也可用于纹理映射，所以我回到这个教程，重新看这两个。

纹理映射是指日可待了。我要告诉您如何创建一个三维物体在今后的教学课程。这将是平面彩色不过这没关系，因为我们将开始纹理映射在下面的教程。

西蒙 iphone-OpenGL ES 教程-05

OpenGL ES 05 – 纹理映射我们的矩形

我已经决定提前介绍纹理映射，因为它可能更容易纹理映射一个对象，而不是面对一个多面（或三维物体） 。此外，似乎这是 iPhone OpenGL ES 的程序员最希望得到的知识，所以我会坚持到现在都说到纹理映射。

我知道我已经跳过了 OpenGL 支持的很多细节，使你直接在屏幕上实验绘制物体，而没有一遍又一遍的介绍 OpenGL 的历史，介绍 OpenGL 和 OpenGL ES 之间有什么不同等等。有时候呢，我也会跳过一些技术细节。

这次，我将介绍相当多的细节，也就是说，这是一个很长的教程。

话虽如此，大部分的代码只是加载纹理到我们的工程里，并且将它放入 OpenGL 的引擎中，这样一来 OpenGL 就可以使用它。这并不复杂，这仅需要调用 iPhone SDK 的一点点工作。

纹理的准备工作

在我们开始使用一个纹理前，我们需要加载它到我们的应用程序里，用 OpenGL 的格式格式化它，并且告诉 OpenGL 在哪里可以找到它。一旦我们做好之前的工作，其他的工作就如同我们之前教程里给矩形上色那么容易。

开启 Xcode，并且打开 EAGLView.h 在编辑区。首先，我们需要提供给 OpenGL 需要的一个变量。增加下面这个声明：

```
GLuint  textures[1];
```

显然，这个一个 GLuint 的数组。你以前见过我使用过 GLfloat，再次说明，GLuint 就是 OpenGL 为无符号整型的一个重命名（typedef）。你应该一直使用 GLxxxx 这样的重命名，而不是使用 Objective_C 的类型参数。因为这些是为了 OpenGL 定义的 OpenGL 命名参数，我们是在使用 OpenGL，而不是开发环境。

下面，我们将调用 OpenGL 函数 glGenTextures()来填充这个变量。我们现在只声明了，之后我们需要覆盖 glGenTextures()函数及这个变量。

在该方法的原型下，添加下列函数：

```
- (void)loadTexture;
```

这里我们将添加代码去加载纹理。

在你的工程里添加上 CoreGraphics Framework

为了加载纹理和对它进行处理，我们将使用 CoreGraphics 框架，因为它提供了所有我们需要的方法，而无需写您在 Windows 的 OpenGL 教程中看到的所有低级别的代码。

在 Xcode “Groups & Files” 侧栏, 右键点击 “Frameworks” 组并选择 Add -> Existing Frameworks...

在搜索框里, 输入 “CoreGraphics.framework” ，并查阅该文件夹中的结果，这些结果符合您的应用目标（ iPhone 的 SDK 2.2.1在我的情况下） 。单击文件夹，并将其添加到您的项目（文件夹图标的框架）。

下一步，我们需要在我们的工程里增加一个纹理图片，所以它需要包含在我们的应用程序包里。下载纹理 checkerplate.png 并且保存它在工程目录里。增加这个图片到你的工程资源目录可以这样做，右键点击资源目录并且选择 Add -> Existing Files... 选择这个图片，这样就添加进入了。

把纹理加载到我们的应用程序和 OpenGL 中
切换到 EAGLView.m 并且我们开始执行 loadTexture 函数.

```
- (void)loadTexture {

}
```

下列的代码将顺序的添加到这个函数中，所以你只要在每行的后面添加就行了。第一件事，我们需要将添加这个图片到我们的应用程序里，使用下列的代码：

```
CGImageRef textureImage = [UIImage imageNamed:@"checkerplate.png"].CGImage;
if (textureImage == nil) {
    NSLog(@"Failed to load texture image");
    return;
```

}

这个 CGImageRef 是 CoreGraphics 的一个数据类型，为了收集图片的所有信息。 要获得此信息，我们要做的就是使用 UIImage 类方法 imageNamed ：创造一个 autorelease'd UIImage 来找到我们应用程序主包的文件名。

To get this information all we do is use the UIImage class method imageNamed: which creates an autorelease'd UIImage finding the file by it's name in our Application's main bundle. UIImage 自动创建的 CGImageRef 和访问的 UIImage 类中的 CGImage 。

现在，我们为了下面的参考，需要获得图片的尺寸.

```
NSInteger texWidth = CGImageGetWidth(textureImage);
NSInteger texHeight = CGImageGetHeight(textureImage);
```

这个 CGImageRef 数据包含了图片的宽和高，但是我们不能直接访问它，我们需要使用上述两种提取函数。

这个 CGImageRef,就象它数据类型名称所暗示的，不包含图片的数据，只说明了图片的数据。所以我们需要开辟一些内存来包含这个图片的数据。

```
GLubyte *textureData = (GLubyte *)malloc(texWidth * texHeight * 4);
```

分配出的正确数据大小，应该是宽*高*4。记得之前的教程里说过，OpenGL 只支持 RGBA 值？每个像素占四个字节，也就是一个 RGBA 种一种颜色占一个字节。

现在，我们需要调用一些绝对重要的函数（张开嘴？？老外真的搞笑）

```
CGContextRef textureContext = CGBitmapContextCreate(
                    textureData,
                    texWidth,
                    texHeight,
```

```
                8, texWidth * 4,
                CGImageGetColorSpace(textureImage),
                kCGImageAlphaPremultipliedLast);

    CGContextDrawImage(textureContext,
                CGRectMake(0.0, 0.0, (float)texWidth, (float)texHeight),
                textureImage);

    CGContextRelease(textureContext);
```

首先，顾名思义，这个 CoreGraphics 函数返回一个 Quartz2D 图形绘制句柄. 基本上来说，我们定义了一个 CoreGraphics 的指针，指向我们的纹理数据，并且告诉它我们纹理的数据和格式。

下面，我们实际上是绘制图片到我们开辟的数据中（纹理数据指针）从我们之前创建的图形绘制句柄的数据指针。这个句柄包含了 OpenGL 需要的所有的信息及数据都复制到 malloc（）中。

当我们完成了 CoreGraphics 的时候，我们需要释放我们创建的 textureContext 句柄。

我知道我加快了上述代码的讲解部分，但是我们对 OpenGL 其他方面的事情更感兴趣。你可以使用这些代码去加载任何被加入你的工程里的 png 格式的图片纹理。

现在，到了 OpenGL 编程了。

现在，还记得我们在头文件中定义的数组了吗？我们现在要使用它。看看下一行代码：

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

我们需要从我们的应用程序复制这个纹理数据到 OpenGL 引擎，所以我们要告诉 OpenGL 为它开辟内存空间。（我们不能直接使用它）. 记得 textures[]定义为 GLuint?一旦我们调用 glGenTextures，OpenGL 就会创建一个句柄或者一个指针，我们加载到 OpenGL 里的每个纹理都是唯一的。OpenGL 返回的值对我们来说并不重要，每次当我们要使用 checkerplate.png

的纹理的时候，我们只需要使用 textures[0]就可以了。OpenGL 就象我们说的那样去做的。

我们也可以一次为多个纹理分配空间。比如，如果我们需要为我们的应用程序准备10个纹理。我们可以如下做：

```
GLuint textures[10];
glGenTextures(10, &textures[0]);
```

在我们的例子里，我们只需要一张纹理，所以我们开辟一张。

下面，我们需要激活我们刚才生成的纹理：

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

第二个参数是显而易见的，它的纹理，我们刚刚建立。第一个参数通常时 GL_TEXTURE_2D 因为所有的 OpenGL ES 在这点上都接受它。 “Full” 的 OpenGL 允许1D 和3D 纹理，但我相信这仍然是需要在今后的 OpenGL ES 的兼容性。

千万别忘记了使用它来激活纹理。

下面，我们发送我们的纹理数据（textureData 的指针）到 OpenGL。OpenGL 在它的那方面（服务面）管理纹理数据，所以数据必须被转换为硬件支持的格式并保存到 OpenGl 的空间里。这个听上去有点拗口，不过大多数的 OpenGL ES 的限制都是相同的。

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, texWidth, texHeight, 0, GL_RGBA, GL_UNSIGNED_BYTE, textureData);
```

遍历这些参数，它们是：

0. •target – 基本上，通常是 GL_TEXTURE_2D
0. •level – 规定纹理的详细程度。0表示允许图片的全部细节。高数字表示 n 级别 mipmap 图片细节。（这边不懂，请知道的朋友告诉我下。）
0. •internal_format – 这个 internal format 和 format 必须是相同的。这两个都是 GL_R

GBA .

0. •width - width of the image

0. •height - height of the image

0. •border – 必须始终设置为0 , OpenGL ES 不支持纹理边界.

0. •format – 必须和 internal_format 相同。

0. •type – 每个像素的类型。想起来没，每个像素为四个字节。因此每个像素占用1个无符号整型（4字节）

0. •pixels – 实际上的图片数据指针。

因此，尽管有很多参数，但大部分都是常识，要需要你输入定义变量 (textureData, texWidth, & texHeight).要记得对你的纹理数据做句柄控制，在 OpenGL 内。

现在，我们已经将数据传输到 OpenGL 里了，我们可以释放我们之前创建的 texturedata.

```
free(textureData);
```

这里有三个函数调用并使用:

```
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glEnable(GL_TEXTURE_2D);
```

这三个函数的调用就是对 OpenGL 做最后的设置，并且设置 OpenGL 的纹理映射的状态。这前两个函数告诉 OpenGL，在放大的(近距离 - GL_TEXTURE_MAG_FILTER) 及缩小的(远距离 - GL_TEXTURE_MIN_FILTER). 时候，如何处理。为了纹理映射的工作及 GL_LINEAR 的选择，你必须至少指定一个。

最后，我们调用 glEnable()告诉 OpenGL 使用纹理，当我们告诉 OpenGL 执行绘制代码的时候。

最后，我们需要在 initWithCoder 初始化里增加这个函数。

```
[self setupView];
[self loadTexture];// Add this line
```

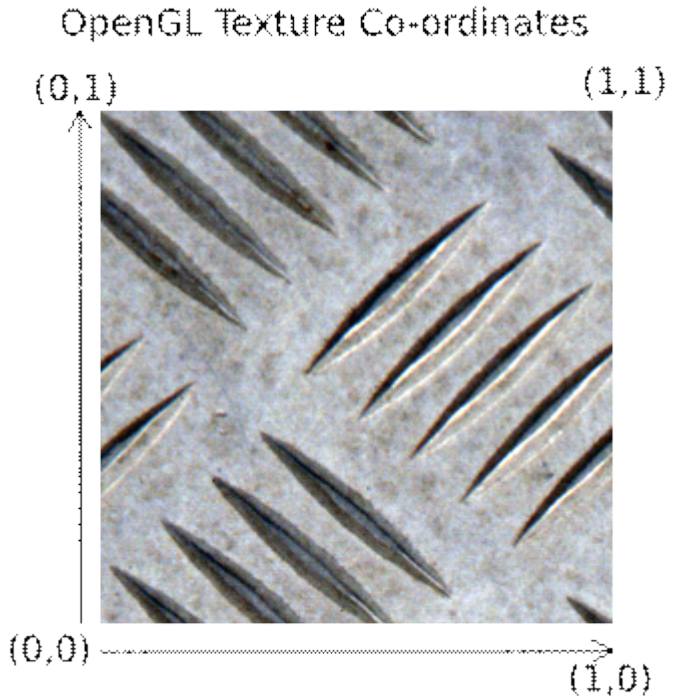
就是添加第二行在 setupView 函数的后面

drawView 的调整

这是一个艰苦的工作。改变 drawView 函数并不比我们之前教程里给矩形上色更困难。首先，注销掉 squareColours[] 数组，我们不需要使用它了。

现在，回忆我们在为矩形上色的时候，我们为每个矩形的顶点，提供了一个颜色的值。当涉及到纹理映射的时候，我们要做同样的事，就象告诉每个顶点是什么颜色一样，我们告诉每个顶点对应的纹理坐标。

在我们这样做之前，我们需要知道什么是纹理坐标。OpenGL 定义纹理坐标的原点(0,0)在左下角，每个轴就是从0—1。看下我们纹理的图的说明：



参照我们的 squareVertices[].

```
const GLfloat squareVertices[] = {
```

```
        -1.0, 1.0, 0.0,                 // Top left
        -1.0, -1.0, 0.0,                // Bottom left
        1.0, -1.0, 0.0,                 // Bottom right
        1.0, 1.0, 0.0                   // Top right
    };
```

你可以看到第一个纹理坐标，我们是不是要指定到左上角的纹理？这里的纹理坐标是(0,1). 我们的第二点是右上方的广场，因此，纹理坐标（ 1 ， 1 ） 。然后，我们去的右下角，这是纹理坐标（ 1 ， 0 ），最后结束的左下角，我们结束纹理坐标（ 0 ， 0 ） 。因此，我们指定 squareTextureCoords [ ]如下：

```
    const GLshort squareTextureCoords[] = {
        0, 1,           // top left
        0, 0,           // bottom left
        1, 0,           // bottom right
        1, 1            // top right
    };
```

注意：我们使用 GLshort 不是 GLfloat 的。添加上述代码到您的项目。

你看看，这个是不是类似于我们的纹理数组？

好了，现在我们需要修改绘制代码。不用管三角形的绘制代码，直接从矩形的代码开始。新的矩形绘制代码如下：

```
    glLoadIdentity();
    glColor4f(1.0, 1.0, 1.0, 1.0);          //NEW
    glTranslatef(1.5, 0.0, -6.0);
    glRotatef(rota, 0.0, 0.0, 1.0);
    glVertexPointer(3, GL_FLOAT, 0, squareVertices);
    glEnableClientState(GL_VERTEX_ARRAY);

    glTexCoordPointer(2, GL_SHORT, 0, squareTextureCoords);        // NEW
```

```
glEnableClientState(GL_TEXTURE_COORD_ARRAY);                    // NEW

glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
glDisableClientState(GL_TEXTURE_COORD_ARRAY);                   // NEW
```

ok,这段代码有四行新的代码。我删除了之前教程里的为矩形上色的代码。第一行代码是调用 glColor4f()，我会在下面的内容中说明。

下面的三行代码你现在应该非常的熟悉了。这里所指的不是物体的顶点或者颜色，我们现在只针对纹理。

```
glTexCoordPointer(2, GL_SHORT, 0, squareTextureCoords);     // NEW
glEnableClientState(GL_TEXTURE_COORD_ARRAY);                // NEW
```

第一个调用就是告诉 OpenGL 我们的纹理坐标数组是在哪里存储并且格式是什么。所不同的，我们告诉它每个坐标只有两个值（这是当然的，因为是2d 纹理），数据类型我们使用的是 GLushort 所以这里用 GL_SHORT，这里没有 stride (0),并且指向我们的坐标指针。

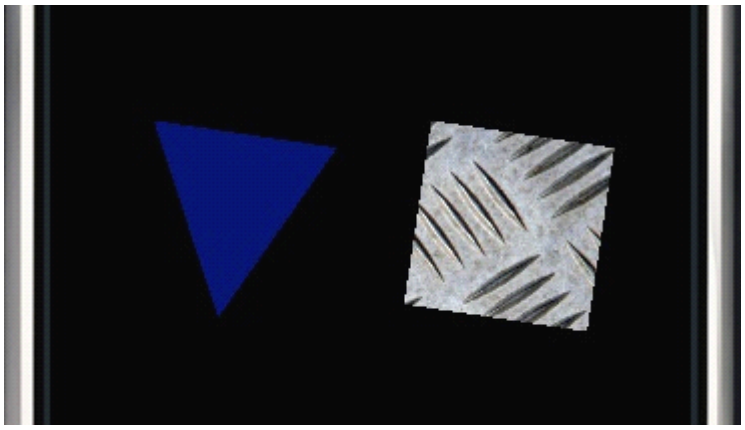
现在我们告诉 OpenGL 客户端状态，为了纹理映射我们指定的坐标数组。

调用 glDrawArrays()没有改变：

```
glDisableClientState(GL_TEXTURE_COORD_ARRAY);                   // NEW
```

记不记得，当我们矩形和三角形采取了不同的颜色的时候，我们关闭了颜色数组？再次，我们需要对纹理映射这样做（关闭它），不然 OpenGL 将使用这个纹理去映射三角形。

保存代码，点击 “Build and Go”, 你可以看到如下的界面



我们的 checkerplate 纹理现在映射到矩形上，而我们的三角形和以前一样。

进一步的实验

首先，让我说下我们增加在绘制矩形之前这行代码：

```
glColor4f(1.0, 1.0, 1.0, 1.0);          // NEW
```

当然，这是改变绘制颜色为白色，不透明。你能猜出来，我为什么要增加这行吗？OpenGL 就是一个状态机，所以一旦我们设置了某个状态，这个状态就一直保留，直到我们改变它。所以绘制颜色被设置为蓝色，直到我们把它改为白色。

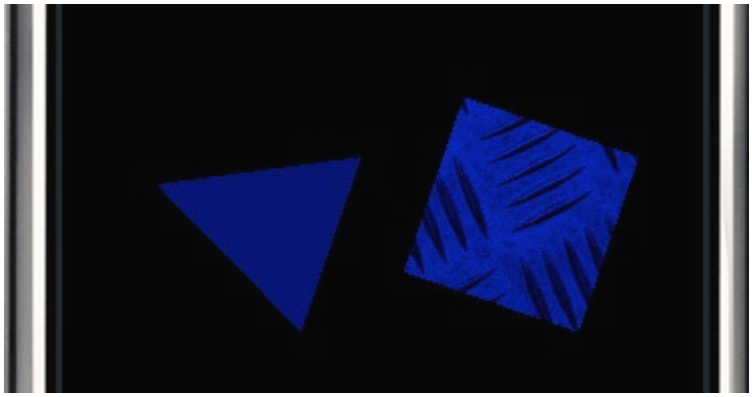
Ok,到纹理映射的时候，OpenGL 将当前颜色设置（蓝色）乘以当前纹理像素做为最后的颜色。这就是

```
                                R      G      B      A
Colour Set:                  0.0, 0.0, 0.8, 1.0
Texture Pixel Colour: 1.0, 1.0, 1.0, 1.0
```

所以，当 OpenGL 执行绘制的，乘积为：

```
Colour_Red * Pixel_Colour_Red = Rendered_colour
0.0        *       1.0                = 0.0
Colour_Green * Pixel_Colour_Green
0.0        *       0.0                = 0.0
Colour_Blue * Pixel_Colour_Blue
0.8        *       1.0                = 0.8
```

注销掉这行 glColour4f 函数，结果就变为了：



当它是白色的时候，乘积就是:

Set Colour : 1.0, 1.0, 1.0, 1.0

is mulitplied by

Pixel Colour : 0.8, 0.8, 0.8, 1.0

Result: 0.8, 0.8, 0.8, 1.0

这就是我们为什么将颜色设置为白色的原因.

好了，就是这样！

在本教程里，我真的说了很多，但我希望你可以看到，实际用于纹理映射的代码不是很多，更多的工作，是在建立纹理的时候。

西蒙 iphone-OpenGL ES 教程-06

OpenGL ES 06 – 3D 坐标里的物体

到目前为止，我们已经对2d 物体做了很好的说明。现在是开始创建3d 物体的时候了。虽然我们不需要太多的改变，它们需要更多的顶点（如果你创建并使用顶点数组）或者更多的坐标转换，如果你想使用多个平面来创建一个立方体。

也许我该先介绍点和线，但是至今我们已经介绍了一些纹理映射矩形及彩色的三角形，我们不必要去研究那些不够有趣的形状！

另外，我们需要回头去看坐标转换，并且介绍关于旋转的更多细节。并且初学者的东西我不需要在解释。那么，这所有的一切都意味着，我还有更多的教程都没有写。

首先，贯穿 drawView 函数
对我们艰难的代码工作说88吧。是时候让 drawView 函数回到最基本的状态了。

将 drawView 函数做成如下：

```
- (void)drawView {

// Our new object definition code goes here

    [EAGLContext setCurrentContext:context];
    glBindFramebufferOES(GL_FRAMEBUFFER_OES, viewFramebuffer);
    glViewport(0, 0, backingWidth, backingHeight);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glMatrixMode(GL_MODELVIEW);

// Our new drawing code goes here

    glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
    [context presentRenderbuffer:GL_RENDERBUFFER_OES];
    [self checkGLError:NO];
```

```
}
```

你应该感谢我之前为3d 空间做的一切，因为我们不需要对深度缓冲做解释或者增加许多新的代码了。这些你已经非常熟悉了。

定义一个3d 物体

我们将产生一个立方体，因为它们构件很容易，是很常见的3d 图形，并且看起来很 cool 的3d 旋转及纹理映射。在我们绘制一个3d 立方体之前，我们需要知道3d 立方体是由6个我们一直在使用的矩形组成的。这很容易，但我将会把你以前的定义都打破。让我们先来定义正面。

```
const GLfloat cubeVertices[] = {

    // Define the front face
    -1.0, 1.0, 1.0,          // top left
    -1.0, -1.0, 1.0,         // bottom left
    1.0, -1.0, 1.0,          // bottom right
    1.0, 1.0, 1.0,           // top right
```

顶面和之前的面几乎完全相同，让我来看看：

```
    // Top face
    -1.0, 1.0, -1.0,         // top left (at rear)
    -1.0, 1.0, 1.0,          // bottom left (at front)
    1.0, 1.0, 1.0,           // bottom right (at front)
    1.0, 1.0, -1.0,          // top right (at rear)
```

请注意，我绘制顶面的方向和我绘制正面的方向不是一样的。（正面是逆时针，而顶面不是）但是我在同一地点开始绘制？如果我们将立方体沿 x 轴旋转90度，那么第一个点在左上，跟下来是右上？

接下来，后面：

```
// Rear face
1.0, 1.0, -1.0,          // top right (when viewed from front)
1.0, -1.0, -1.0,         // bottom right
-1.0, -1.0, -1.0,        // bottom left
-1.0, 1.0, -1.0,         // top left
```

注意到顶点的出发点了吗？我们将以这样的方式完成其他面。

```
// Bottom Face
-1.0, -1.0, 1.0,         // Bottom left front
1.0, -1.0, 1.0,          // right front
1.0, -1.0, -1.0,         // right rear
-1.0, -1.0, -1.0,        // left rear
```

逆可以看到，我用同样的方法和起点。在你的头脑里面想像如何旋转这些面，看到点排列的方式。

最后，我们完成左面和右面:

```
// Left face
-1.0, 1.0, -1.0,         // top left
-1.0, 1.0, 1.0,          // top right
-1.0, -1.0, 1.0,         // bottom right
-1.0, -1.0, -1.0,        // bottom left

// Right face
1.0, 1.0, 1.0,           // top left
1.0, 1.0, -1.0,          // top right
1.0, -1.0, -1.0,         // right
1.0, -1.0, 1.0           // left
```

这里是立方体的完整定义:

```
const GLfloat cubeVertices[] = {

    // Define the front face
    -1.0, 1.0, 1.0,          // top left
    -1.0, -1.0, 1.0,         // bottom left
    1.0, -1.0, 1.0,          // bottom right
    1.0, 1.0, 1.0,           // top right

    // Top face
    -1.0, 1.0, -1.0,         // top left (at rear)
    -1.0, 1.0, 1.0,          // bottom left (at front)
    1.0, 1.0, 1.0,           // bottom right (at front)
    1.0, 1.0, -1.0,          // top right (at rear)

    // Rear face
    1.0, 1.0, -1.0,          // top right (when viewed from front)
    1.0, -1.0, -1.0,         // bottom right
    -1.0, -1.0, -1.0,        // bottom left
    -1.0, 1.0, -1.0,         // top left

    // bottom face
    -1.0, -1.0, 1.0,
    -1.0, -1.0, -1.0,
    1.0, -1.0, -1.0,
    1.0, -1.0, 1.0,

    // left face
    -1.0, 1.0, -1.0,
    -1.0, 1.0, 1.0,
    -1.0, -1.0, 1.0,
    -1.0, -1.0, -1.0,
```

```
        // right face
        1.0, 1.0, 1.0,
        1.0, 1.0, -1.0,
        1.0, -1.0, -1.0,
        1.0, -1.0, 1.0
    };
```

如果你对坐标系统有问题，你最好在你的脑海里去想像它。如果你的思维还停留在2d 平面上，那你真的要努力了，我们已经开始3d 了。

将 cubeVertices 放到 drawView 函数，并且说明 “New object definition goes here”.

OK, 现在我们要开始绘制这个棒棒糖（不是翻译错误）。

绘制立方体

最简单的办法就是你用你以前看过的代码直接去绘制立方体。不过现在，我们要使用一些先进的（你理解了，它就很简单）的方法来绘制3d 物体。然而，现在，让我来介绍如何在3d 中绘制图形。

我们开始的代码不需要在解释了。在我们的注释之下，添加如下代码：

```
glLoadIdentity();
glTranslatef(0.0, 0.0, -6.0);
glVertexPointer(3, GL_FLOAT, 0, cubeVertices);
glEnableClientState(GL_VERTEX_ARRAY);
```

这里没有任何新的代码。我们在开始的重置了我们的矩阵，移动我们的立方体到屏幕里以让我们可以看到，告诉 OpenGL 将要使用的顶点数组的格式及数据存储位置。

后面的代码几乎和你之前用过的相同:

```
// Draw the front face in Red
glColor4f(1.0, 0.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
```

这里没有新的代码，我们告诉 OpenGL 将红色作为绘制颜色，提取从0开始的4个顶点来绘制一个矩形。现在，用我们数组里面的下4个顶点来绘制顶面。

```
// Draw the top face in green
glColor4f(0.0, 1.0, 0.0, 1.0);
glDrawArrays(GL_TRIANGLE_FAN, 4, 4);
```

看看这里的 glDrawArrays()函数. 如果你还记得我对你描述过的，我说，第二个参数是数据开始的偏移。是的，因为我们是绘制立方体的第二个面，我们需要告诉 OpenGL 从要偏移4个值（也就是 cubeVertices[4], 0-3是第一个面），然后我们用之后的4个顶点来绘制。（老外的教程说的真仔细，我看过3d 很多本书了，这是第一次了解第二个参数的意义）

现在，我们来绘制后面：

```
    // Draw the rear face in Blue
    glColor4f(0.0, 0.0, 1.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 8, 4);
```

同样的定义，我们从 cubeVertices[8]开始绘制. 用同样的方法来绘制其他三个面：

```
    // Draw the bottom face
    glColor4f(1.0, 1.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 12, 4);

    // Draw the left face
    glColor4f(0.0, 1.0, 1.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 16, 4);
```

```
// Draw the right face
glColor4f(1.0, 0.0, 1.0, 1.0);
glDrawArrays(GL_TRIANGLE_FAN, 20, 4);
```

我们改变了每个面的颜色，并且为 glDrawArrays ()改变了偏移值。

现在，如果你点击了“Build and Go”，你只会得到一个红色的矩形。为了看到所有的6个面，让我们来一起旋转三个轴。

在 glLoadIdentity()之前，添加下面的代码。

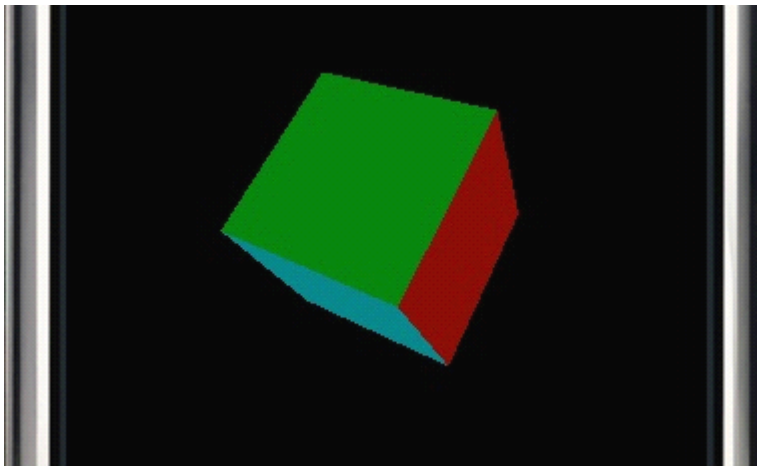
```
rota += 0.5;
```

我们的老朋友 rota 先生回来了。（这作者在搞笑，不太高明）。现在我们需要其他的老朋友 glRotatef() 。在 glTranslatef()函数之后，添加下面：

```
glRotatef(rota, 1.0, 1.0, 1.0);
```

以前，我们只需要使用 glRotatef() 来旋转一个轴，现在我们需要三轴同时旋转。

现在我们点击 “Build and Go” 来看看你得到的：



欢迎你，3d 物体。

如何对它进行纹理映射？

我认为，我们现在需要的不只是一个纯色的物体的。让我们使用上个教程里的纹理来纹理映射六个面来使得这个立方体更加的精彩。

好了，我们保持我们工程里面的纹理加载函数。我们只需要修改我们的 drawView 函数。现在，我带你们去看看如何一次性的快速纹理映射。

首先，你还记得这个吗，在上个教程里的

```
const GLshort squareTextureCoords[] = {
    // Front face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right
```

恩，我们可以很容易的纹理映射一个面。我们需要扩充它。然而，这是很容易的。想起来我在确定立方体的各个面是如何使用一样的排列方式的吗？（顺时针规范顶点）现在我们来找到这是为什么。

当 OpenGL 绘制一个纹理到立方体的一个面的时候，因为我们绘制每个面的时候都使用了偏移（4，8，或者12），纹理映射的时候也会采用相同的偏移。所以，为了纹理映射六个面，我们需要重复之前的4个坐标5遍。

```
const GLshort squareTextureCoords[] = {
        // Front face
        0, 1,       // top left
        0, 0,       // bottom left
        1, 0,       // bottom right
        1, 1,       // top right
```

```
    // Top face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right

    // Rear face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right

    // Bottom face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right

    // Left face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right

    // Right face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right
};
```

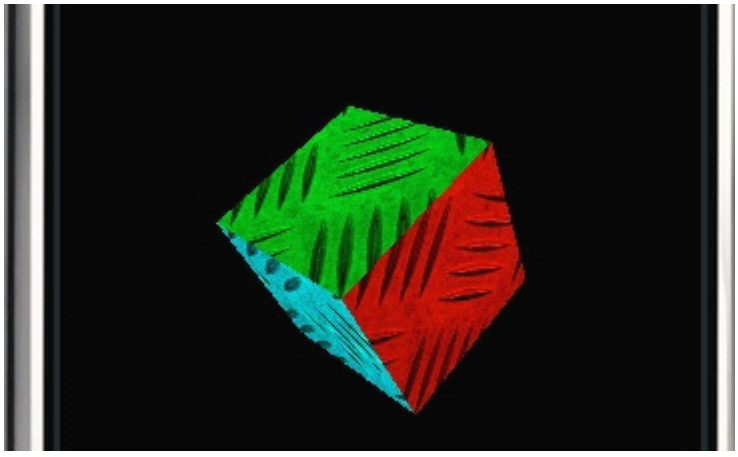
数据的剪切和粘贴工作完成了！

现在，我们只需要使用几行代码来纹理映射我们的立方体。

在绘制第一个面之前，添加下面代码：

```
glTexCoordPointer(2, GL_SHORT, 0, squareTextureCoords);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

这是很简单，别删除颜色处理那段。看看新的生成结果；



嘿，这就象魔法一样。一个纹理映射过的3d 物体在3d 空间中旋转。

在纹理映射图片中的一个概念

在上个教程里，我忘记了说明这个概念了。虽然我建议你使用自己的纹理来进行纹理映射，但是纹理的尺寸必须是2的 幂数。图片的宽和高必须是 1, 2, 4, ... 32, ... 512, 1024.这个宽和高不相同也可以，但必须是2的 幂数。所以32 x 512同64 x 64一样是正确的尺寸。而30 x 30不是。

glRotatef()和你物体顶点的一个概念

同样，我鼓励你创建你自己的物体。你有没有注意到，我一直把0，0，0作为我三角形，矩形及立方体的中心？这是因为当我们旋转的时候，OpenGL 将会以0，0，0作为模型矩阵的

中心。它不会调整自己的模型到0，0，0在旋转。如果你的物体不是以0，0，0为中心来创建的，你旋转的时候就知道什么叫“lop-sided”.

西蒙 iphone-OpenGL ES 教程-07

OpenGL ES 07 – 对物体进行独立的转换

起初，我想下一章介绍光照的，但是我又想到别的。我们依然有很多基本对象及转换没有完成，特别是我们在不同的坐标系中对不同的物体进行转换。

记得我们如何在整个屏幕内使用 glTranslatef() 和 glRotatef ()？如果我想作一些不同的事情，我就会使用一个很方便的函数： glLoadIdentity().但是 glLoadIdentity()是“代价昂贵”在屏幕渲染技术中，所以现在，我会采用更有效的方法。

要作到这点，我将介绍一个新的对象。我们将增加一个金字塔，并且使用 glTranslatef()来移动它，使用 glRotatef()来旋转它。和立方体独立开，并且不使用 glLoadIdentity() 来重置矩阵。

在我们的世界里增加一个金字塔
这不是仅仅增加一个三角形，让我们坚持使用3d 物体，我们会增加一个金字塔（想像一下 Giza 平原上的大石头）

有件事，我在以前的教程中避免去说明的，就是我们创建的立方体事一个复杂对象，即物体是由一个以上的原点组成的。从技术上来说，我们的矩形也是一个复杂物体，但我们只使用

了单一的 OpenGL 的函数去绘制它，我们可以将其看为一个简单物体，我避免说明复杂物体是因为我不想让我们的工作看起来是一个“艰苦的工作”

现在你已经完成了一个复杂物体了。恭喜你！并且，现在是时候建立第二个了。

金字塔不是难事。它是由一个矩形以及会聚到这个矩形上面的中心点的4个三角形组成的。一旦你将物体可以分解为你所认知的简单图元，任何复杂的物体都不是问题。唯一的变化是你物体中图元的数目而已。

好，开启 Xcode 并且开启上个教程的工程。这次不要删除任何东西，我们增加代码来构成一个平行线。

为了帮助你组成金字塔，我们需要分解它的各个组成部分，让我们开始：

```
const GLfloat pyramidVertices[] = {
    // Our pyramid consists of 4 triangles and a square base.
    // We'll start with the square base
    -1.0, -1.0, 1.0,          // front left of base
    1.0, -1.0, 1.0,           // front right of base
    1.0, -1.0, -1.0,          // rear left of base
    -1.0, -1.0, -1.0,         // rear right of base
```

因此，我们创建了一个新的物体，和我们作的一样。这个矩形是底座，所有的 y 坐标都是-1.0。

现在，我们可以创建金字塔的正面，这次我们创建的三角形而不是矩形。

```
    // Front face
    -1.0, -1.0, 1.0,          // bottom left of triangle
    1.0, -1.0, 1.0,           // bottom right
    0.0, 1.0, 0.0,            // top centre -- all triangle vertices
                              //      will meet here
```

这是我们创建的唯一的真实三角形，这个三角形的角在矩形的前边，然后向后倾斜，上面的点在矩形的中心上面（在 x 轴线上）

我们继续创建其他的三个三角形，使用相同的方法，下面是完整的顶点数组。

```
// Our new object definition code goes here
    const GLfloat pyramidVertices[] = {
        // Our pyramid consists of 4 triangles and a square base.
        // We'll start with the square base
        -1.0, -1.0, 1.0,            // front left of base
        1.0, -1.0, 1.0,             // front right of base
        1.0, -1.0, -1.0,            // rear left of base
        -1.0, -1.0, -1.0,           // rear right of base

        // Front face
        -1.0, -1.0, 1.0,            // bottom left of triangle
        1.0, -1.0, 1.0,             // bottom right
        0.0, 1.0, 0.0,              // top centre -- all triangle vertices
                                    //       will meet here

        // Rear face
        1.0, -1.0, -1.0,    // bottom right (when viewed through front face)
        -1.0, -1.0, -1.0,           // bottom left
        0.0, 1.0, 0.0,              // top centre

        // left face
        -1.0, -1.0, -1.0,           // bottom rear
        -1.0, -1.0, 1.0,            // bottom front
        0.0, 1.0, 0.0,              // top centre

        // right face
        1.0, -1.0, 1.0,             // bottom front
```

```
        1.0, -1.0, -1.0,            // bottom rear
        0.0, 1.0, 0.0               // top centre
    };
```

增加金字塔的定义到 drawView 方法里和 cubeVertices[] 的定义在一起。

在我们继续前，我想先说明下金字塔的定义。

首先，这是第一个我们同时提供了三角形及矩形的对象。这个金字塔是由一个矩形和4个三角形组成的。因为我们调用 glDrawArrays()为每个单独的图元，但这不防碍我们每次使用单独的定义来绘制不同的图元。那么，当我们绘制金字塔，一切都变的清楚了。

第2，再一次的提醒。我已指定所有的顶点是逆时针的。即使后面看起来似乎是顺时针的，但是从 OpenGL 的角度看，它还是逆时针的。

其实，我想在我3d 图形教学结束前，我们会一直讨论这个问题。

绘制金字塔

现在低头去找立方体的代码。首先，删除 glLoadIdentity() 这行,它已经不需要了。

在 rota+=0.5以后，我们增加绘制金字塔的代码。现在，我们要在不影响立方体的情况下，使用 glTranslatef() 去移动金字塔，并且使用 glRotatef() 去旋转金字塔。为了做到这点，我们需要在不影响其他图元的情况下使用 glTranslatef() 和 glRotatef().

OpenGL 为我们使用这两个函数提供了快捷的方法。

```
    glPushMatrix();

    // Translation and drawing code goes here....
```

```
glPopMatrix();
```

OpenGL 告诉我们如果要将我们的矩阵变换放如到堆栈中就要使用 glPushMatrix().我将我们的 pyramidVertices[] 和 cubeVertices[] 目标 “推” 到 OpenGL’s 的堆栈中.

Note: 如果你不明白我说的堆栈是什么意思,那么你可能需要一本 c 或者 object c 的工具书。

所以，我们的数据复制是安全的，我们可以象其他 OpenGL 大师一样进行转换了。让我们开始绘制这个金字塔吧！

```
glPushMatrix();
{
    glTranslatef(-2.0, 0.0, -8.0);
    glRotatef(rota, 1.0, 0.0, 0.0);
    glVertexPointer(3, GL_FLOAT, 0, pyramidVertices);
    glEnableClientState(GL_VERTEX_ARRAY);
```

首先要指出的是， glPushMatrix()后面的那个{.这个不是必要的，但是你在学习的时候最好加上它，{}内表示了被推入堆栈和弹出堆栈之间做的工作。

在 glPushMatrix() 后面的四行，我马上解释给你听. 所有我们所做的就是要求 glTranslatef（ ）把金字塔远离（ 0 ， 0 ， 0 ）的左侧和回8点到屏幕（进一步远离观众） 。那么我们的金字塔周围旋转的 X 轴，而不是所有三个轴计算的立方体的例子在过去的教程。最后，我们只要告诉 OpenGL 的有关数据，并使它能够被使用。

好了，在我们做好坐标转换以后，我们开始绘制金字塔。

```
    // Draw the pyramid
    // Draw the base -- it's a square remember
    glColor4f(1.0, 0.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
```

前四个坐标表示了我们金字塔的底部。我们将它的颜色设置为红色（也可以进行纹理映射），并且使用 GL_TRIANGLE_FAN 来绘制矩形。我们是从(pyramidVerticies[0~3]),使用4个的顶点。

Ok,我们完成矩形了，现在开始绘制第一个正面的三角形。

```
        // Front Face
        glColor4f(0.0, 1.0, 0.0, 1.0);
        glDrawArrays(GL_TRIANGLES, 4, 3);
```

除里改变颜色，我们改变绘制模式为 GL_TRIANGLES 的原因是显然的，我们已经从数组的第4个(pyramidVerticies[4~6])开始,延伸3个顶点来绘制。所以，你可以看到，一个矩形和一个三角形可以很容易的共存于一个数据结构里。

下一步，我们继续绘制我们其他三个三角形：

```
        // Rear Face
        glColor4f(0.0, 0.0, 1.0, 1.0);
        glDrawArrays(GL_TRIANGLES, 7, 3);

        // Right Face
        glColor4f(1.0, 1.0, 0.0, 1.0);
        glDrawArrays(GL_TRIANGLES, 10, 3);

        // Left Face
        glColor4f(1.0, 0.0, 1.0, 1.0);
        glDrawArrays(GL_TRIANGLES, 13, 3);
    }
    glPopMatrix();
```

每次，我们只是改变颜色，并改变开始的偏移值。OpenGL 知道每个顶点是由三个坐标组成的，通过我们的 glVertexPointer() 函数。

最后，我们关闭}，并且调用 glPopMatrix().

现在，在这点上我们可以绘制立方体。但是，请注意，你可以一遍又一遍的重复绘制，只需要将绘制函数放在 glPushMatrix() 和 glPopMatrix()中间.

绘制立方体－修订

这里是绘制立方体的完整代码。最大的改变就是代码头尾的函数 glPushMatrix() 和 glPop Matrix()

```
glPushMatrix();
{
    glTranslatef(2.0, 0.0, -8.0);
    glRotatef(rota, 1.0, 1.0, 1.0);
    glVertexPointer(3, GL_FLOAT, 0, cubeVertices);
    glEnableClientState(GL_VERTEX_ARRAY);

    // Draw the front face in Red
    glColor4f(1.0, 0.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);

    // Draw the top face in green
    glColor4f(0.0, 1.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 4, 4);

    // Draw the rear face in Blue
    glColor4f(0.0, 0.0, 1.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 8, 4);

    // Draw the bottom face
    glColor4f(1.0, 1.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 12, 4);
```

```
        // Draw the left face
        glColor4f(0.0, 1.0, 1.0, 1.0);
        glDrawArrays(GL_TRIANGLE_FAN, 16, 4);

        // Draw the right face
        glColor4f(1.0, 0.0, 1.0, 1.0);
        glDrawArrays(GL_TRIANGLE_FAN, 20, 4);
    }
    glPopMatrix();
```

其他的变化只有 glTranslatef 的参数。

这里是完整的 drawView 函数。

```
- (void)drawView {

    // Our new object definition code goes here
    const GLfloat pyramidVertices[] = {
        // Our pyramid consists of 4 triangles and a square base.
        // We'll start with the square base
        -1.0, -1.0, 1.0,            // front left of base
        1.0, -1.0, 1.0,             // front right of base
        1.0, -1.0, -1.0,            // rear left of base
        -1.0, -1.0, -1.0,           // rear right of base

        // Front face
        -1.0, -1.0, 1.0,            // bottom left of triangle
        1.0, -1.0, 1.0,             // bottom right
        0.0, 1.0, 0.0,              // top centre -- all triangle vertices
                                    //       will meet here
```

```
    // Rear face
    1.0, -1.0, -1.0,      // bottom right (when viewed through front face)
    -1.0, -1.0, -1.0,           // bottom left
    0.0, 1.0, 0.0,              // top centre

    // left face
    -1.0, -1.0, -1.0,           // bottom rear
    -1.0, -1.0, 1.0,            // bottom front
    0.0, 1.0, 0.0,              // top centre

    // right face
    1.0, -1.0, 1.0,             // bottom front
    1.0, -1.0, -1.0,            // bottom rear
    0.0, 1.0, 0.0               // top centre
};

const GLfloat cubeVertices[] = {

    // Define the front face
    -1.0, 1.0, 1.0,             // top left
    -1.0, -1.0, 1.0,            // bottom left
    1.0, -1.0, 1.0,             // bottom right
    1.0, 1.0, 1.0,              // top right

    // Top face
    -1.0, 1.0, -1.0,            // top left (at rear)
    -1.0, 1.0, 1.0,             // bottom left (at front)
    1.0, 1.0, 1.0,              // bottom right (at front)
    1.0, 1.0, -1.0,             // top right (at rear)

    // Rear face
    1.0, 1.0, -1.0,             // top right (when viewed from front)
```

```
    1.0, -1.0, -1.0,        // bottom right
    -1.0, -1.0, -1.0,       // bottom left
    -1.0, 1.0, -1.0,        // top left

    // bottom face
    -1.0, -1.0, 1.0,
    -1.0, -1.0, -1.0,
    1.0, -1.0, -1.0,
    1.0, -1.0, 1.0,

    // left face
    -1.0, 1.0, -1.0,
    -1.0, 1.0, 1.0,
    -1.0, -1.0, 1.0,
    -1.0, -1.0, -1.0,

    // right face
    1.0, 1.0, 1.0,
    1.0, 1.0, -1.0,
    1.0, -1.0, -1.0,
    1.0, -1.0, 1.0
};

const GLshort squareTextureCoords[] = {
    // Front face
    0, 1,       // top left
    0, 0,       // bottom left
    1, 0,       // bottom right
    1, 1,       // top right

    // Top face
    0, 1,       // top left
    0, 0,       // bottom left
```

```
        1, 0,       // bottom right
        1, 1,       // top right

        // Rear face
        0, 1,       // top left
        0, 0,       // bottom left
        1, 0,       // bottom right
        1, 1,       // top right

        // Bottom face
        0, 1,       // top left
        0, 0,       // bottom left
        1, 0,       // bottom right
        1, 1,       // top right

        // Left face
        0, 1,       // top left
        0, 0,       // bottom left
        1, 0,       // bottom right
        1, 1,       // top right

        // Right face
        0, 1,       // top left
        0, 0,       // bottom left
        1, 0,       // bottom right
        1, 1,       // top right
};

[EAGLContext setCurrentContext:context];
glBindFramebufferOES(GL_FRAMEBUFFER_OES, viewFramebuffer);
glViewport(0, 0, backingWidth, backingHeight);
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glMatrixMode(GL_MODELVIEW);
```

```
glTexCoordPointer(2, GL_SHORT, 0, squareTextureCoords);
glEnableClientState(GL_TEXTURE_COORD_ARRAY);

// Our new drawing code goes here
rota += 0.5;

glPushMatrix();
{
    glTranslatef(-2.0, 0.0, -8.0);
    glRotatef(rota, 1.0, 0.0, 0.0);
    glVertexPointer(3, GL_FLOAT, 0, pyramidVertices);
    glEnableClientState(GL_VERTEX_ARRAY);

    // Draw the pyramid
    // Draw the base -- it's a square remember
    glColor4f(1.0, 0.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);

    // Front Face
    glColor4f(0.0, 1.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLES, 4, 3);

    // Rear Face
    glColor4f(0.0, 0.0, 1.0, 1.0);
    glDrawArrays(GL_TRIANGLES, 7, 3);

    // Right Face
    glColor4f(1.0, 1.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLES, 10, 3);

    // Left Face
    glColor4f(1.0, 0.0, 1.0, 1.0);
```

```
    glDrawArrays(GL_TRIANGLES, 13, 3);
}
glPopMatrix();

glPushMatrix();
{
    glTranslatef(2.0, 0.0, -8.0);
    glRotatef(rota, 1.0, 1.0, 1.0);
    glVertexPointer(3, GL_FLOAT, 0, cubeVertices);
    glEnableClientState(GL_VERTEX_ARRAY);

    // Draw the front face in Red
    glColor4f(1.0, 0.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);

    // Draw the top face in green
    glColor4f(0.0, 1.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 4, 4);

    // Draw the rear face in Blue
    glColor4f(0.0, 0.0, 1.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 8, 4);

    // Draw the bottom face
    glColor4f(1.0, 1.0, 0.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 12, 4);

    // Draw the left face
    glColor4f(0.0, 1.0, 1.0, 1.0);
    glDrawArrays(GL_TRIANGLE_FAN, 16, 4);

    // Draw the right face
    glColor4f(1.0, 0.0, 1.0, 1.0);
```

```
        glDrawArrays(GL_TRIANGLE_FAN, 20, 4);
    }
    glPopMatrix();



    glBindRenderbufferOES(GL_RENDERBUFFER_OES, viewRenderbuffer);
    [context presentRenderbuffer:GL_RENDERBUFFER_OES];

    [self checkGLError:NO];
}
```

请注意，我们并没有为金字塔添加一个新的纹理坐标数组。我们可以使用相同的坐标数组，因为它依然为这个范例工作。

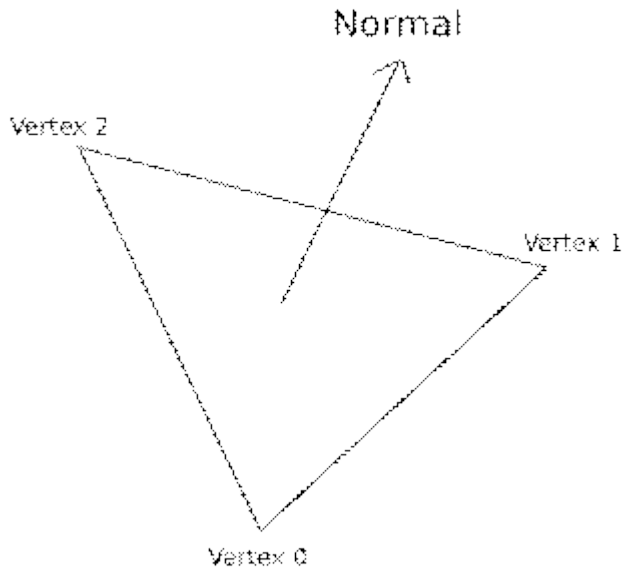
改变好了以后，点击”Build an Go”

你所有需要主要的是，这两个物体正在独立的改变自己。金字塔只沿 x 轴旋转，而立方体沿3轴旋转。再次说明，颜色只是让你更容易的分辨每个面。

采用3d 概念：法线

记不记得我早些时候说过，要保证每个三角形都是逆时针的，即使有些背面的三角形从正面看起来象是顺时针的？

恩，这是因为你要保证你每个面的数据都是“标准”的。简单的说，一个面的法线是一个假想的线从物体的面抽出。为了说明这点，请看下面的图：



在上面的图片里，这个三角形的法线就是一条从面出来的线。当前的三角形的法线表示，这个三角形的3个顶点是逆时针的。我会在介绍光照的那章里说明什么是法线。

（在这里我真的要说说，我真的很佩服这教程的原作者，他居然可以将如此简单的代码说的如此的复杂，请注意，我没有一点的贬义，而是对他非常的敬佩。明明每个教程只有几行代码，他确不怕麻烦的一次又一次的说明其中的概念，解释每一行代码的表面意思和内部的概念。这样细致的工作打个比方来说，就是一个博士生在细心的教幼儿园小朋友唱歌。不是一般人可以做到的。比我们所阅读到的其他 OpenGL 书籍更加的认真，也更加的细致。这也是我继续翻译下去的主要动力。）

西蒙 iphone-OpenGL ES 教程-09

OpenGL ES 09 –不使用 Mr. Buzzy 的混合色 1

请注意：我对我写这个教程时的状态不是很满意，所以如果你有些地方有疑惑，请不要在意。如果哪里错了，也别担心，我会在第2节里继续解释它。

谁是 Mr. Buzzy? 那是我的厨房搅拌机！（英文里搅拌机和混合色是同样的）我们当然不是在谈论我的厨房搅拌机或者混合3d 软件，我们在说 OpenGL 的混合。然而，这个部分可能要说2，3个教程。影响混合的效果是一些很小的细节，这就需要很多的说明。在以后的教程中会有详细的说明。

我还记得我第一使用混合时候的代码。幸运的是，我没有写自己的代码，但效果真的很酷。那是一个 SGL 的显卡。非常强大的处理能力，比不上现在的苹果，但当时，真的是很壮观的。

最主要的原因是因为有一个激动人心的时刻，在当时，我们还没有人见过混合时执行纹理映射，同时还有灯光及坐标转换。

无论如何，让我们把学习融合。混合，是相结合的过程中两个图像一起，使图像显示的前端部分透明。例如，只说你有一块红色的有机玻璃，并期待通过。世界上会出现红色的，因为颜色的有机玻璃将改变颜色的所有对象，通过它您可以查看。

因此，使混合工作，我们需要一个对象在前台是部分透明的（即阿尔法值小于1.0的 RGBA 颜色定义）和一些背后的，将被“混合型”的对象。

这是混合的关键，你需要两个对象，前面的对象是透明的，透明的部分即刻看到后面的物体。

我应该告诉你们，混合的正式名称是“阿尔法合成”。 我不会要求可以认为（从来没有），但您可能会看到这个词了。计算机图形学上涨了很长的路，因为我研究过，但正式的时候，我学过这个，我认为，融合是一种形式的阿尔法合成和 Alpha 合成的一种方法是单一或联合起草的图像和 Alpha 通道。

使用入门

下载第7章的教程: AppleCoder-OpenGLES-07.zip

我们将使用这个工程，并使用纹理映射过的金字塔和立方体在背景上，为了显示混合的效果。打开 Xcode。

在 OpenGL 中的混合

为了混合在 OpenGL 中工作起来，我们需要改变 OpenGL 中的“状态”以让我们可以使用混合。我敢肯定你知道如何做到这点。

```
glEnable(GL_BLEND);
```

添加到 drawView 函数，我们绘制之前，这样就可以使用半透明了，关闭这个状态使用 glDisable()。

好了，混合接通（记得要关掉它，如果你不需要它） 。一个伟大的事情 OpenGL 的是，可以执行的混合使用不同的混合方法来产生不同的混合效果。这些所谓的混合功能。我就第一个节和讨论他们在下面详细说明。

好，回到 drawView 函数，我们要定义一个矩形，来放到我们两个物体之前。

```
const GLfloat blendRectangle[] = {
    1.0, 1.0, -2.0,
    -1.0, 1.0, -2.0,
    -1.0, -1.0, -2.0,
```

```
        1.0,  -1.0,  -2.0
    };
```

我们想使两个。因此，在借鉴代码将使用 glPushMatrix （ ）和 glPopMatrix （ ）一对把它们独立。

向下，到绘制金字塔和立方体的方法那里。为了正确的显示混合效果，我们必须先绘制一个不透明的物体。然后在绘制透明的。就像一个老画家一样。

现在，考虑下 OpenGL 的当前状态。我们当前有纹理映射，我们要这个纹理去做图吗？不是的，所以我们要关闭纹理映射。

```
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
```

然后，开始混合：

```
    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE,  GL_ONE);
```

glEnable 打开混合。光混合是不够的，我们要告诉 OpenGL 混合是如何工作的，否则默认状态下，不会做任何事情。 这就是 glBlendFunc （ ）发挥作用。不必担心这个，现在我要讨论的混合功能的详细阐述;现在，让我们刚刚得到的工作，然后我们可以实验。

恩，纹理关闭了，混合打开了，混合模式也提交给 OpenGL 了，现在是时候绘制两个矩形了。

```
    glPushMatrix();
    {
        glTranslatef(0.0,  1.0,  -4.0);
        glVertexPointer(3,  GL_FLOAT,  0,  blendRectangle);
        glEnableClientState(GL_VERTEX_ARRAY);
        glColor4f(1.0,  0.0,  0.0,  0.4);
        glDrawArrays(GL_TRIANGLE_FAN,  0,  4);
```

```
}
glPopMatrix();

glPushMatrix();
{
    glTranslatef(0.0, -1.0, -4.0);
    glColor4f(1.0, 1.0, 0.0, 0.4);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
}
glPopMatrix();
glDisable(GL_BLEND);
```

除了调用 glDisable （ ）在结束时，没有什么新的这里。我们仅仅是我们的 blendRectangle 定义，定位它在显示器上，颜色，并提请它。然后，我们需要禁用 OpenGL 的融合，否则将结合我们的金字塔和立方体下次将使用此功能（您可以尝试以后如果你愿意的话） 。

那就是了，点击“Build and Go” :

注意颜色的金字塔和立方体的转变，他们跨越两个颜色的矩形？这是融合工作。颜色与两位前矩形是部分透明的（ 40 ％透明的），结合或混合的颜色的物体，以创建一个新的色彩。

因此，黄色的矩形看起来比红色的矩形亮。

如果你只得到了一个红色及黄色的矩形，那么有可能你绘制函数的地方插错了位置。我们需要绘制透明物体在不透明物体之后。或者你改变混合模式。

这是它最基本的混合。没有什么特别的，但是我们了解了混合的部分。

在继续之前的快速小结
因此，使用混合的时候，我们需要一个透明的对象在我们屏幕的前面。我们先绘制实心的对象，这时候我们可以调用 glEnable()，使用 glBlendFunc()来设置混合，最后从后到前得到我们转换过的对象。

现在我们需要进入混合真正复杂的地方。设置混合因子。

混合因子
我刚刚删除了我刚刚写的内容。这个可以决定像素的最终颜色，基本上是数学矩阵（尽管它并没有被提到）。不，我不认为是一个正确的事情。

对我来说是时候了，让你得到更多的创造性和使用性的东西，但是你需要第一个理论：混合是怎么工作的。

如我先前所说的，你需要一个半透明的物体，并且有个背景上的东西进行混合工作。你也可以简单的和背景混合，不过我们每次都使用 glClear()来清除了背景，所以你可能什么都得不到。

所以，在显示器上得每个像素，OpenGL 都找到两件事：

1.这是源像素。这是我们当前绘制得新像素，从部分透明得对象（在红色和黄色矩形）

2. 这是目标像素。这是我们当前显存中得像素。这个就是正在绘制得半透明物体。在我们例子里，这个是一个混合对象，包含了实心对象（金字塔及立方体）和背景。

3.这是源像素混合因子指示符。这个是 glBlendFunc()的第一个参数。它告诉 OpenGL 如何处理这个新的像素。

4.这个是目标像素混合因子指示符。这个是 glBlendFunc()的第二个参数，并且告诉 OpenGL 如何处理目标像素。

找到上述四个信息以后。OpenGL 可以计算出新的源及目标的混合因子，进而放入目标内存中。简单的说，我只是删除了所有的垃圾。

钥匙是 glBlendFunc()中的两个参数。

在上面的例子里，我们使用了下面的代码设置混合功能：

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

这两个参数都是相同的: GL_ONE. 第一个参数表示了源或者传入像素的值，第二个参数表示了目标像素的值。

这里有一系列不同的混合因子说明符可以传入 glBlendFunc()。这里需要你知道的是，它们单独并没有作用，必须要结合使用才可以表示我们的混合效果。

这是最重要的一点。我知道我有没有列出的选择，但是我需要你首先记住这一条理论。

如果你想获得最佳的混合效果，你需要明确你的想法，因为这里，新的颜色与部分透明的对象（我们的矩形）和原来的实心的对象是相互影响的。这是两者的结合混合因素控制而成的效果。

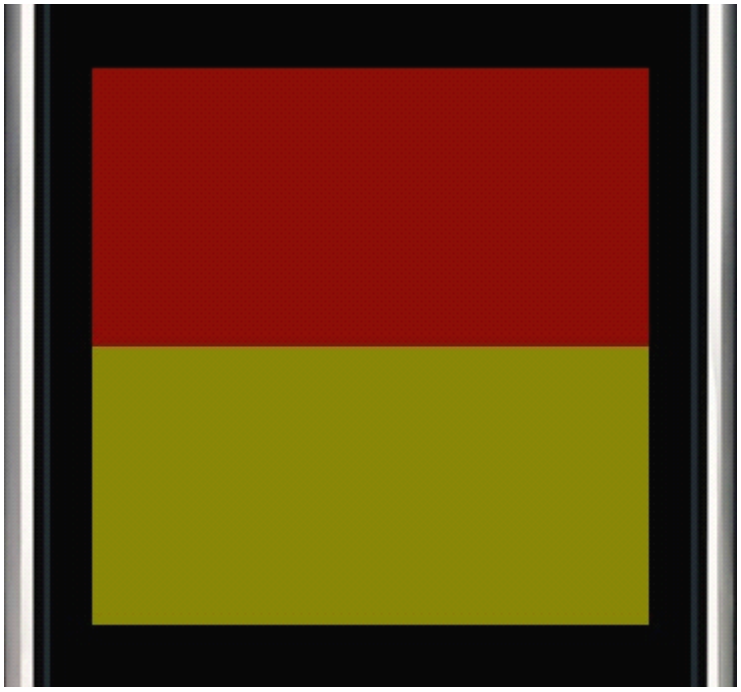
要明白这点，我们可以通过一些例子去了解。这两个参数到底表示什么意思并不重要！我想告诉你，当你改变参数的时候，发生了什么。

这个函数，默认打开了混合。

glEnable():

glBlendFunc(GL_ONE, GL_ZERO);

我们将第二个参数改为 GL_ZERO. 记得吗，第二个参数是影响到我们在缓冲区的实心的物体的，而不是部分透明物体，它使我们有了如下的效果：



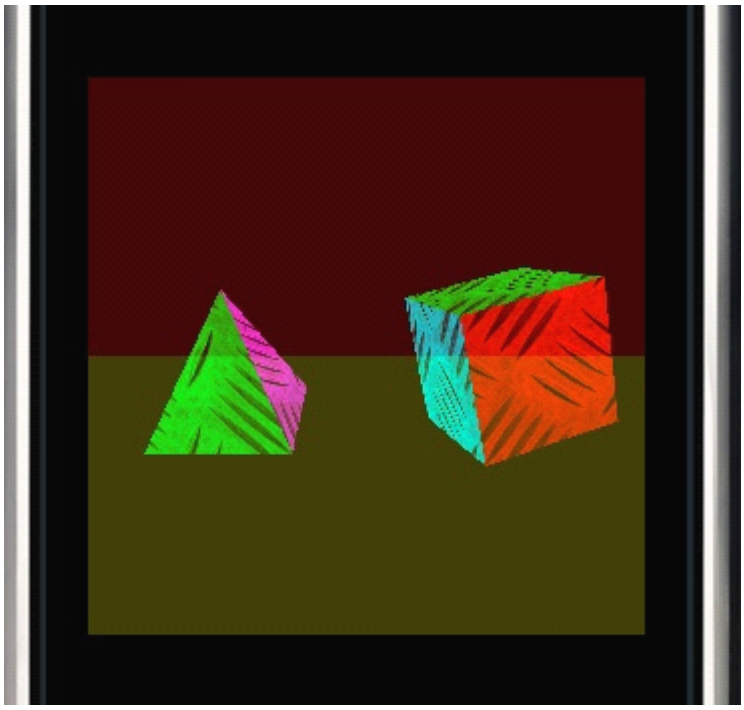
哦，我们的背景物体发生了什么？不要太多解释，你知道 GL_ZERO 是最重要的线索。

这不是因为“没有混合”，而是因为目标像素被乘以了 RGBA 颜色(0,0,0,0),导致没有任何渲染。这个源（两个矩形）因为是 GL_ONE 所以被乘以了(1,1,1,1).这个像素就被完全的显示出来。

如果你没有对 glBlendFunc()做任何修改，这是和现在是相同的，因为这是默认的状态。

再次改变两个参数位下面：

glBlendFunc(GL_ZERO, GL_ONE);



不用怀疑，那两个矩形消失的（这可是你做的，你把它们乘以了0）

好了，你已经知道了 GL_ZERO 和 GL_ONE 表示的是什么意思了。现在我们看看 glBlendFunc()中支持的参数列表：

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

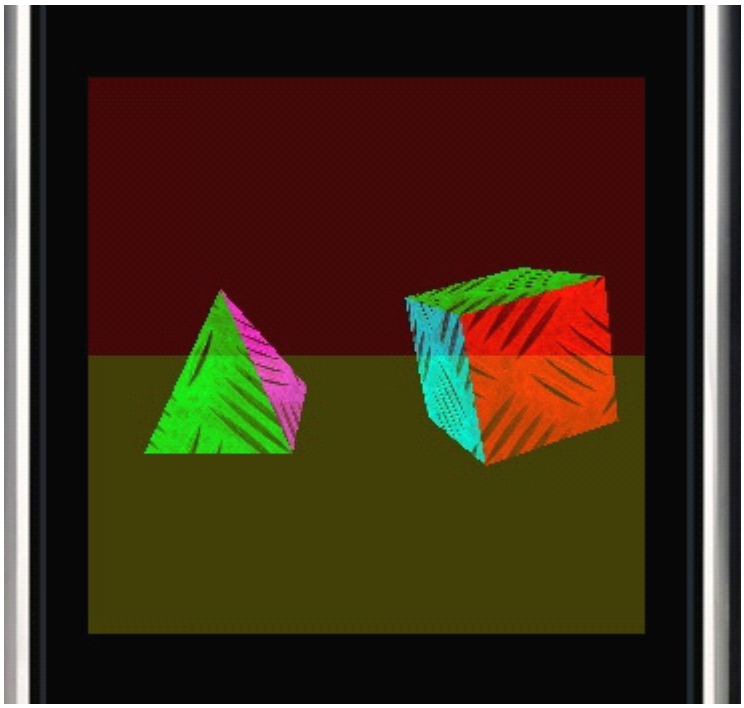
GL_SRC_ALPHA_SATURATE

我会在之后的教程中详细的去解释它。感兴趣的朋友可以自己去网上查阅。

混合的例子

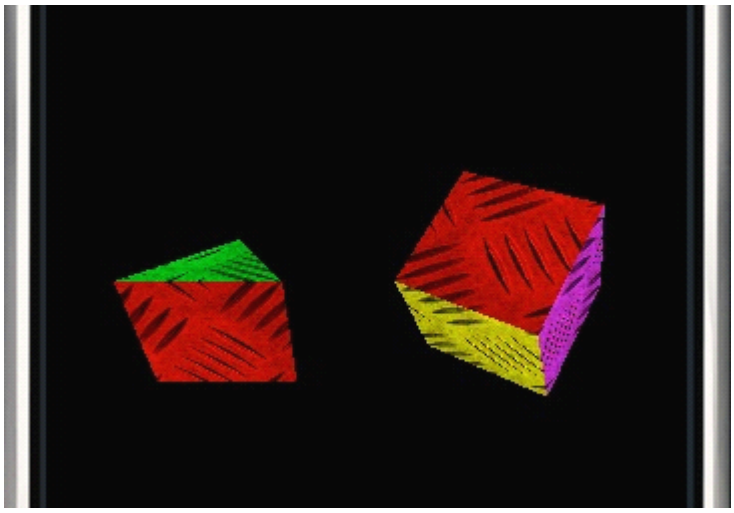
我也通过几个例子来说明这些参数。只是为了让你了解不同的混合效果。

glBlendFunc(GL_SRC_ALPHA, GL_ONE);



请注意，前面的两个矩形更加的透明了？似乎实心的物体更亮了，而透明矩形更暗了，我们没做什么事情，是你的眼睛欺骗了你。

glBlendFunc(GL_ONE_MINUS_DST_ALPHA, GL_ONE);



我们的矩形消失了？恩，不正确。它们的 alpha 是目标的－1。所以我们的目标的 alpha 原来是1.0 。所以 1.0 -1.0 ＝ 0.0

需要注意的一件事是：这个背景颜色被 glClearColor 设置，alpha 是1.0。那么我们修改 clear 颜色。

```
glClearColor(0.0, 0.0, 0.0, 0.0);
```

只改变这行，我们看看效果:

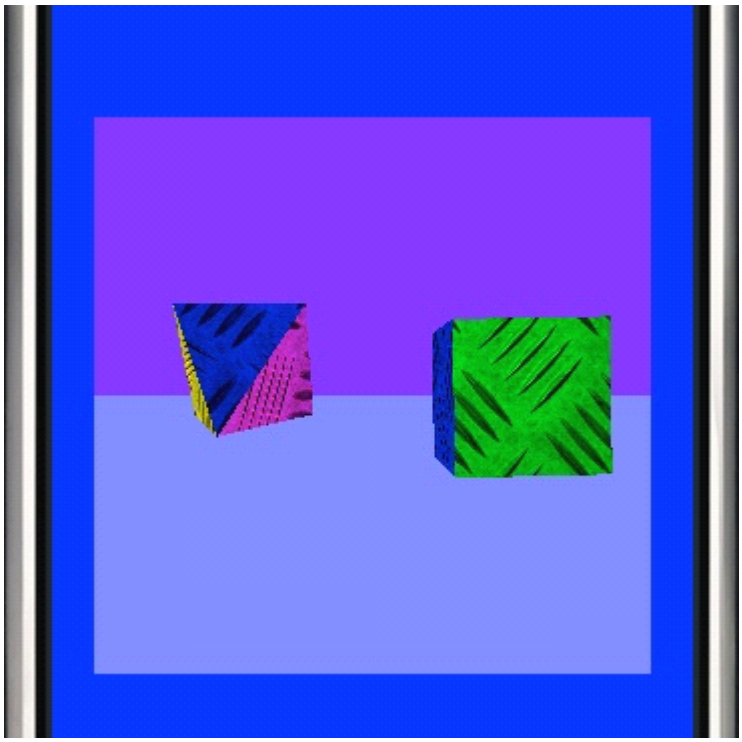
你能看出这是怎么一回事吗？虽然看起来移动到了实心物体的背景，但这真的是混合在工作。从前面的例子知道，无论 OpenGL 在渲染那一个像素，其中只要有一个 alpha 为1.0，这个矩形就是100％透明了。现在背景有一个 alpha 为0.0，所以1.0-0.0＝1.0。

让我们来看看有没有更接近的。

我们红色矩形的颜色定义是: (1.0, 0.0, 0.0, 0.4). 所以如果背景是0 alpha, 这个渲染或混合的颜色就是 (1.0, 0.0, 0.0, 0.4). 背景颜色是黑色是最简单的，让我们来把它改为蓝色。

```
glClearColor(0.0, 0.0, 1.0, 0.0);
```

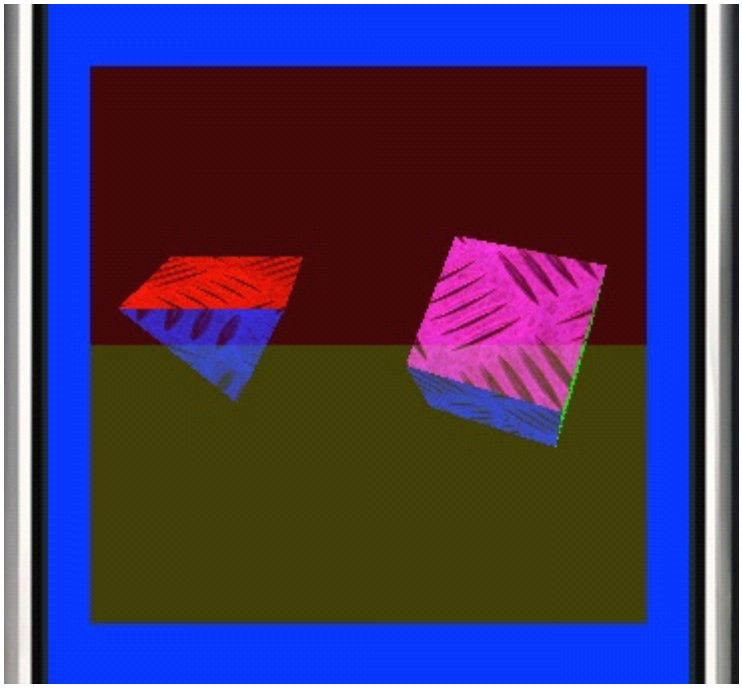
我们将得到如下的图片:



蓝色的背景是显然的。现在立方体及金字塔的混合如前一样。但是透明矩形已经发生了变化，看起来像是灰色的。

我将通过最后一个例子及计算。告诉大家如何得到最后的像素的。

glBlendFunc(GL_SRC_ALPHA, GL_DST_ALPHA);

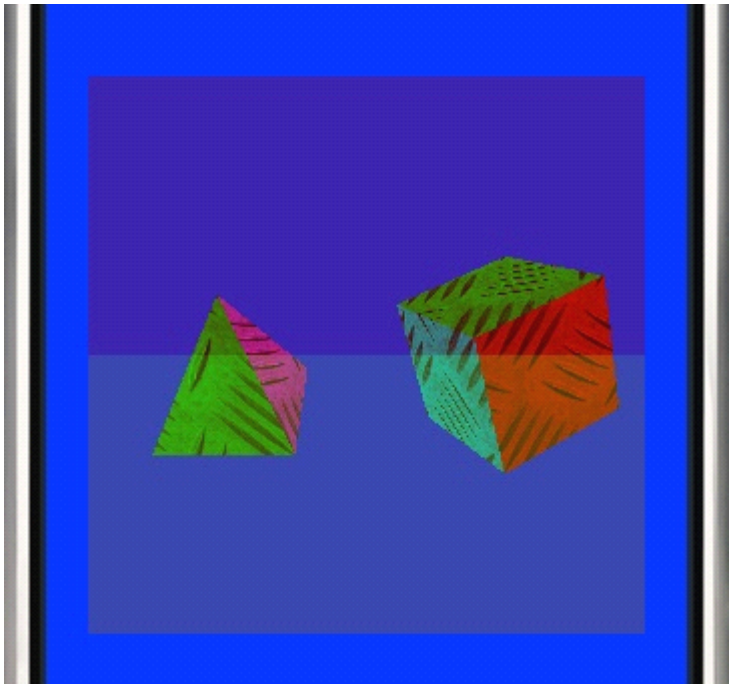


最后一个混合组合….

让我们看看什么是最常见的参数配对：

glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

在我们这的测试结果，看起来像如下：

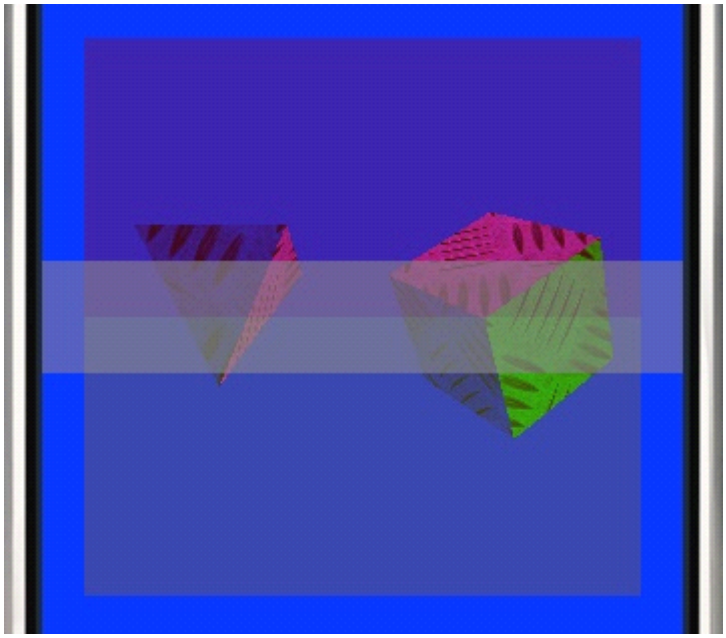


添加额外的混合

你不需要为一个对象的每个像素设置混合。你可以在部分透明的物体上跌加几层。添加下面的函数，在绘制第二个矩形之后。

```
glPushMatrix();
{
    glTranslatef(0.0, 0.0, -3.0);
    glScalef(1.0, 0.3, 1.0);
    glColor4f(1.0, 1.0, 1.0, 0.6);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
}
glPopMatrix();
```

我们要做的是增加了第三个矩形，OpenGL 将混合到一起



当然，我做了一个还没介绍的函数: glScalef().

Scaling – 一些我还没有介绍的东西!
Scaling 就是为对象进行放大缩小, 这里有三个参数: xscale, yscale, and zscale.

在上面的例子中, xscale 和 zscale 是1。所以保持原尺寸。大于1就是放大，小于1就是缩小。上面是0.3，那么就是缩小到原来的30％。立方体和金字塔就在双层混合后的结果。

很简单。它应该包括之前的处理。

西蒙 iphone-OpenGL ES 教程-10

OpenGL ES 10 – 多重纹理，重复纹理，并结束书本时代

我的朋友昨晚留在我家，并且我们喝点啤酒一起聊天。这时候我告诉他我正在写一些教程。所有的时间都是我在说，他的脸上不断的出现这是什么什么的表情，我告诉他，所有人都在网上看到这些教程。他不赞成这个：把所有的东西都放在网上实在是太奇怪了。
我自己在想，现在已经不是书本的时代了，我们可以在网上看到许多好的教程，来帮助我们学习。是时候结束书本时代了。

（伟大的共享精神万岁!!!）

本教程需要使用的纹理在这里下载: Tutorial10Textures.zip

改变 loadTexture[]
第一件需要做的是就是改变 loadTexture[]. 基本上，我们可以改变调用如下：

```
- (void)loadTexture:(NSString *)name intoLocation:(GLuint)location;
```

向前看。我们改变 loadTexture[] 函数让它知道加载纹理的名词和路径就可以加载纹理. 现在改变 loadTexture[] 函数如下:

```
- (void)loadTexture:(NSString *)name intoLocation:(GLuint)location {

    CGImageRef textureImage = [UIImage imageNamed:name].CGImage;
```

不要忘记删除 @"checkerplate.png" 取而带之的是我们的文件名. 删除 glGenTextures() 函数并且改变 glBindTexture() 函数如下：

```
glBindTexture(GL_TEXTURE_2D, location);
```

就是这样，现在这个方法更加的有用，我们将调用 glGenTextures() 在我们的 initWithCoder[] 函数里。当我们要加载某个纹理的时候，我们就调用 loadTexture[] ,在本教程里，我们将加载6个纹理, 所以在 initWithCoder[] 里调用完 setupView, 我们增加如下的函数:

```
        [self setupView];
        glGenTextures(6, &textures[0]);
        [self loadTexture:@"bamboo.png" intoLocation:textures[0]];
        [self loadTexture:@"flowers.png" intoLocation:textures[1]];
        [self loadTexture:@"grass.png" intoLocation:textures[2]];
        [self loadTexture:@"lino.png" intoLocation:textures[3]];
        [self loadTexture:@"metal.png" intoLocation:textures[4]];
        [self loadTexture:@"schematic.png" intoLocation:textures[5]];
```

另外，切换到头文件，改变纹理如下：

```
GLuint textures[6];
```

获得渲染

好，回到 drawView[]. 删除有关金字塔的所有东西。我们已经不需要再使用它了。删除我们使用的颜色。所以看起来就如下：

```
    // Our new drawing code goes here
    rota += 0.2;

    glPushMatrix();
    {
        glTranslatef(0.0, 0.0, -4.0);          // Change this line
        glRotatef(rota, 1.0, 1.0, 1.0);
        glVertexPointer(3, GL_FLOAT, 0, cubeVertices);
        glEnableClientState(GL_VERTEX_ARRAY);

        // Draw the front face
        glDrawArrays(GL_TRIANGLE_FAN, 0, 4);

        // Draw the top face
        glDrawArrays(GL_TRIANGLE_FAN, 4, 4);

        // Draw the rear face
        glDrawArrays(GL_TRIANGLE_FAN, 8, 4);

        // Draw the bottom face
        glDrawArrays(GL_TRIANGLE_FAN, 12, 4);

        // Draw the left face
        glDrawArrays(GL_TRIANGLE_FAN, 16, 4);
```

```
        // Draw the right face
        glDrawArrays(GL_TRIANGLE_FAN, 20, 4);
    }
    glPopMatrix();
```

你可以不需要使用 glPushMatrix() 和 glPopMatrix() .如果你想你的代码100％的优化，就可以删除它。

现在我们已经获得了一些改变，让我们第二次停下来，思考 OpenGL 的状态。

好了，记得刚才我们加载过纹理，我们可以使用 glBindTexture()来告诉 OpenGL 我们希望那个纹理工作。所以，再加载完6个纹理以后，OpenGL 将默认使用最后一个加载的纹理，如果你点击 “Build and Go”, 你将看到一个只有单一纹理的立方体。

因为我们想每个面都有一个纹理，我们需要告诉 OpenGL 我们想使用那个纹理来绘制，在我们调用 glDrawArrays()之前。所以，我们的纹理代码如下：

```
    glPushMatrix();
    {
        glTranslatef(0.0, 0.0, -4.0);
        glRotatef(rota, 1.0, 1.0, 1.0);
        glVertexPointer(3, GL_FLOAT, 0, cubeVertices);
        glEnableClientState(GL_VERTEX_ARRAY);

        // Draw the front face
        glBindTexture(GL_TEXTURE_2D, textures[0]);
        glDrawArrays(GL_TRIANGLE_FAN, 0, 4);

        // Draw the top face
        glBindTexture(GL_TEXTURE_2D, textures[1]);
        glDrawArrays(GL_TRIANGLE_FAN, 4, 4);
```

```
        // Draw the rear face
        glBindTexture(GL_TEXTURE_2D, textures[2]);
        glDrawArrays(GL_TRIANGLE_FAN, 8, 4);

        // Draw the bottom face
        glBindTexture(GL_TEXTURE_2D, textures[3]);
        glDrawArrays(GL_TRIANGLE_FAN, 12, 4);

        // Draw the left face
        glBindTexture(GL_TEXTURE_2D, textures[4]);
        glDrawArrays(GL_TRIANGLE_FAN, 16, 4);

        // Draw the right face
        glBindTexture(GL_TEXTURE_2D, textures[5]);
        glDrawArrays(GL_TRIANGLE_FAN, 20, 4);
    }
    glPopMatrix();
```

点击 “Build and Go”

回答一个问题，那些代码完成了如上的功能。

纹理包装

下一节就是我一直在说的重复和包装纹理。重复纹理只是简单的多次重复相同的纹理。这是非常有用的内容，用较小的纹理组成一面墙。（必须是要整齐的纹理）

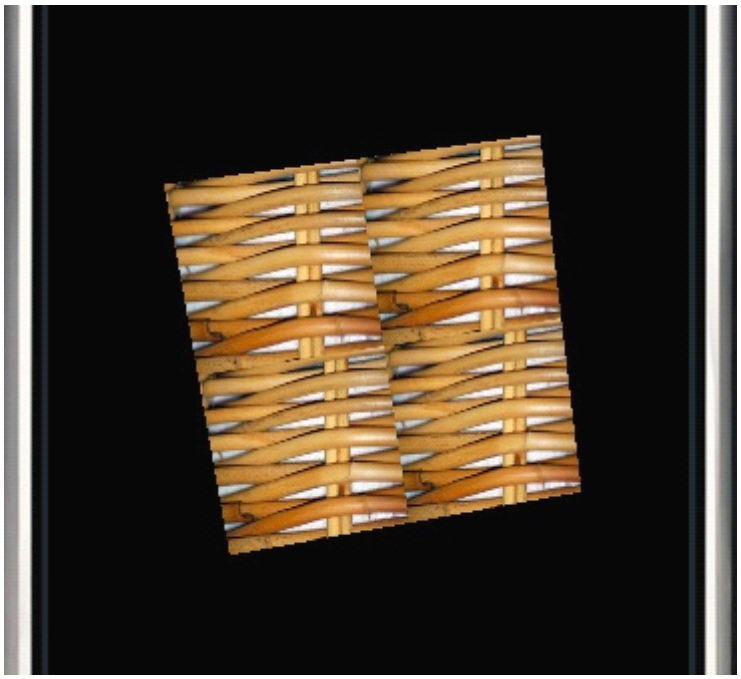
我们要做的是绘制我们的立方体的一个面，使用一个纹理重复四次。我要用这个竹子纹理改变正面，你可以做你任意喜欢的一个面。

这非常容易，你还记得 OpenGL 是如何从图片中抽取像素去渲染对象的吗？它使用了纹理坐标数组，之前我都告诉你，使用0.0-1.0。如果我要使图像出现两次，我之要将1.0改变为2.0，重复的纹理就会出现了。

所以，在我们的纹理数组里，改变如下:

```
const GLshort squareTextureCoords[] = {
        // Front face
        0, 2,        // top left
        0, 0,        // bottom left
        2, 0,        // bottom right
        2, 2,        // top right

        // Top face
        0, 1,        // top left
        0, 0,        // bottom left
        1, 0,        // bottom right
        1, 1,        // top right
```

点击 “Build and Go”:

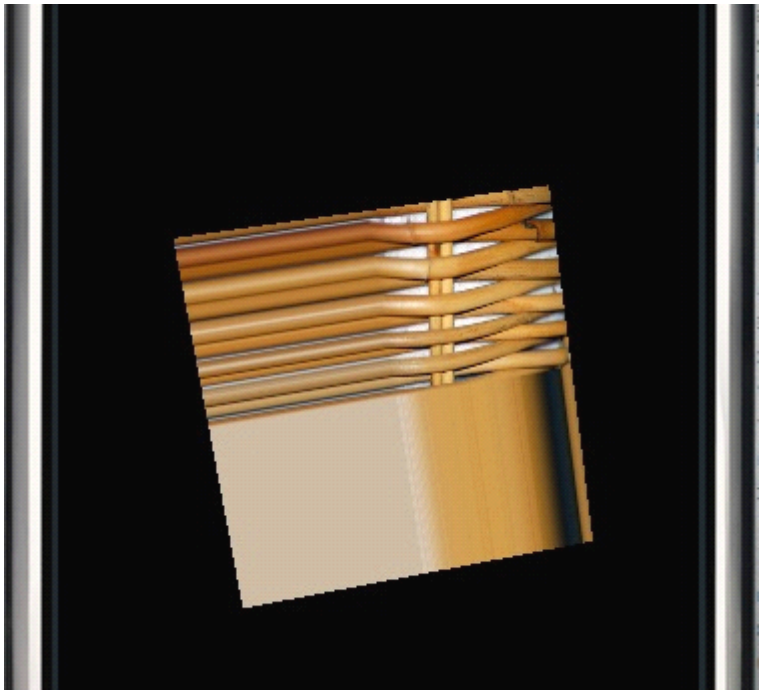


我只是修改了前面，如果你喜欢，你可以把其他面都做修改。

这是 OpenGL 默认的纹理映射状态，当纹理坐标数组里的值大于1.0的时候。如果你想改变这个状态，你也可以调用 glTexParameterf().比如，你可以改变如下：

```
        // Draw the front face
    glBindTexture(GL_TEXTURE_2D, textures[0]);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
    glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
```

上面的代码，在加载纹理坐标中数组大于1以后，就会出现如下效果



纹理的边缘像素一直延伸到左上角。这就是 GL_CLAMP_TO_EDGE.

让我快速解释这两行代码：

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);

我们调用了 glTexParameterf() 当我们加载纹理的时候设置纹理过滤。像一般的 OpenGL 函数，它具有“双重功能”。有三个参数，但是在 OpenGL ES 只有两个有效果。第一个参数一般使 GL_TEXTURE_2D ,OpenGL ES 只支持它。

第二个参数是告诉 OpenGL 什么“变量”是我们要改变的，一般来说，是 GL_TEXTURE_WRAP_x.

这第三个参数就是我们要设置的。

OpenGL 通过设置 “变量” or “设置” GL_TEXTURE_WRAP_S t 到 GL_CLAMP_TO_EDGE,

我们告诉 OpenGL 不要重复纹理而是延伸 s 坐标 (什么是 “s”, 后面会说). 我们也做了相同的 t 坐标，等下运行下模拟器，看看是什么样的。

这里设置的默认值是 GL_REPEAT.

s t 坐标?
不，不是玩笑。OpenGL 在它的纹理中使用了 s t 坐标系统。它确实存在，以避免混淆，因为你必须记住我们的纹理并不是真实存在于我们的3d 世界，他们只是我们放置到3d 世界的某些东西。所以，你可以认为 s 是横向，t 是纵向.但你旋转它的时候，它是不是一样的使用？

西蒙 iphone-OpenGL ES 教程-11

OpenGL ES 11 – 单纹理，多视图，纹理渲染，以及数学灵感

我在高中的最后一年，我正在做一个先进的数学题目，是因为有趣，看着油漆干燥。有不少次我都没有谈到，因为我的那些先进的教学理念在课堂的环境根本没有激励我。

有一天，我继续坐在教室的后面发呆，突然我发现白板上的这样一行：

[a b c] . [d e f]

我坐在笔直。我知道这是矩阵数学，这是我一直暴露在阅读有关图形程序，但我还没有完全理解。嗯，我有兴趣再次，鼓舞注意，甚至喜欢自己。

兴趣是一个有趣的事情。我提到这一点，因为无论我如何努力，我不能完全获得该教程混合权利。所以，我刚刚张贴，并决定回来后，我所有的“灵感” 。

现实将是我永远也不会得到启发寻找在立方体和金字塔，试图处理日益复杂的 OpenGL ES 的概念。其上市时间开始（缓慢）增加的复杂性场面在我们正在尝试英寸这样，灵感不应枯竭。

增加了现场的复杂性将沿着放缓，但我们不能作出任何巨大的跳跃在任何时间，以免造成人民后面。但是，增加的复杂性，我们将和您的结果将学习现场管理，以及在 OpenGL ES 的概念。

今天的教程将刚才我们在隧道。但是，在我们达到现场，我们有一些其他位做家务。首先，我们需要支付最后一次专题载入纹理中的 iPhone ，但直到去年我离开，因为它是最重要的（阅读：我忘记了它） 。然后我们要引起我们的隧道使用4种不同的材质，但加载从一个单一的纹理文件。最后，我们会做一些基本呈现的纹理，触摸基地再次与融合。

iPhone’s Y != OpenGL’s Y
OpenGL’s Y 不适合很多的平台，包括 iPhone 的。所以当我们使用渲染的时候，y 是从上往下的。

为了解决这个问题，不要调用 glRotatef() ,虽然它可以解决这个问题，但是会浪费 CPU/GPU 的时间。

这是仅于的解决办法。

0. 1. 改变纹理之前捆绑在一个图像编辑器，以便在加载的正确方式。这一工程，但有可能减少跨平台的兼容性。还需要更多的工作。我不会做这件事。
0. 2.改变纹理的坐标数组为右坐标。也就是说，左下角将是(0,1).OpenGL 并不关系你的纹理数据，它只需要知道每个顶点对应的纹理坐标就可以了。
0.
3。改变 loadTexture[] 函数，当它加载图片的时候自行翻转。但是这样一来，你每次加载图片都要浪费时间。并且增加 CPU/GPU 的消耗。

“最佳方案”是使用第3点。

我将改变 loadTexture[] 函数。在本教程里，一切还很好，还没有 OpenGL 不能做的事。

唯一的改变事修改下面的函数。

```
CGContextRef textureContext = CGBitmapContextCreate(textureData,
                                                    texWidth, texHeight,
                                                    8, texWidth * 4,
                                                    CGImageGetColorSpace(textureImage),
                                                    kCGImageAlphaPremultipliedLast);

// Rotate the image -- These two lines are new
CGContextTranslateCTM(textureContext, 0, texHeight);
CGContextScaleCTM(textureContext, 1.0, -1.0);
```

这为我们的图像指定了正确的方向。

从单个纹理中出来的多个纹理

做为今天的乐趣，我们将使用单一的纹理来生成4个纹理对象：两个墙的纹理，地板的纹理和天花板的纹理。

在上个教程里，我们为立方体的每个面都加载了一个纹理。我们每一次加载纹理并传送给 OpenGL，它都会在 OpenGL 的控制空间里增加一个副本。因此，我们可以使用 free()去释放内存，在 loadTexture 里使用 malloc()为纹理图片开启内存。显然，我们每次改变纹理都需要时间，每次使用新的纹理都要通知 OpenGL 绑定新的纹理。

这个方法给了你一些变革，但它不是真正的性能。虽然是一个方便的技巧，我只是想告诉你可以使用纹理的一个部分。

第一件事就是创建一个空白的图片，大小是512*512，足够放下4张256*256的图片。然后我们复制每个图片到这个大图片里，保存为 png 文件。

至少有些工作你可以自己完成了，我将分离出每个纹理通过控制我的纹理坐标。

在我们定义纹理坐标前，我们需要定位我们使用的顶点坐标。我们需要一个矩形如下：

```
const GLfloat elementVerticies[] = {
    -1.0, 1.0, 0.0,       // Top left
    -1.0, -1.0, 0.0,      // Bottom left
    1.0, -1.0, 0.0,       // Bottom right
    1.0, 1.0, 0.0         // Top right
};
```

ok,如果我们想绘制木材纹理（左下角的），我们需要指定下面的纹理坐标。

```
const GLfloat combinedTextureCoordinate[] = {
    // The wood wall texture
    0.0, 1.0,       // Vertex[0~2] top left of square
    0.0, 0.5,       // Vertex[3~5] bottom left of square
    0.5, 0.5,       // Vertex[6~8] bottom right of square
    0.5, 1.0,       // Vertex[9~11] top right of square
```

和一起一样，他们就是坐标。每个纹理坐标对应一个顶点坐标。

从底部到木材纹理的顶部，正好是全部大小的一半， y 坐标将是0.5而不是以前的0.0。以前，我们是使用的 GLshort，现在我们使用 GLfloats。

同样，x 方向也是到达0.5就截至了。

这里是四个纹理的坐标：

```
const GLfloat combinedTextureCoordinate[] = {
    // The wood wall texture
    0.0, 1.0,       // Vertex[0~2] top left of square
    0.0, 0.5,       // Vertex[3~5] bottom left of square
    0.5, 0.5,       // Vertex[6~8] bottom right of square
    0.5, 1.0,       // Vertex[9~11] top right of square

    // The brick texture
    0.5, 1.0,
    0.5, 0.5,
    1.0, 0.5,
    1.0, 1.0,

    // Floor texture
    0.0, 0.5,
    0.0, 0.0,
    0.5, 0.0,
```

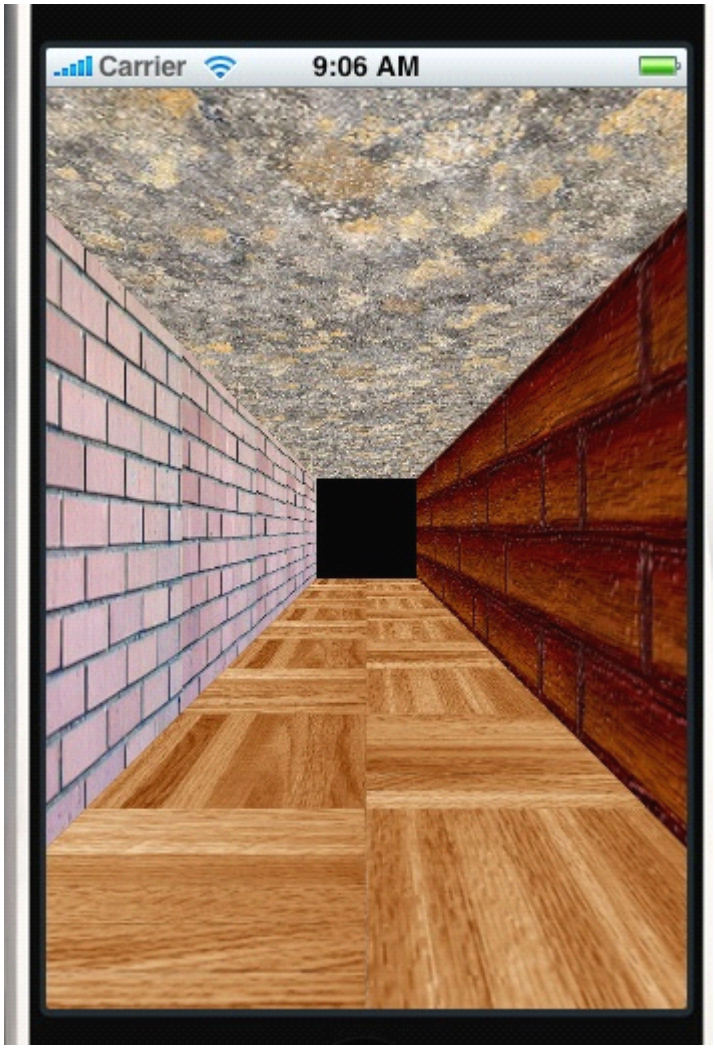
```
        0.5,  0.5,

        // Ceiling texture
        0.5,  0.5,
        0.5,  0.0,
        1.0,  0.0,
        1.0,  0.5
    };
```

最后一件事，我定义了一些宏来让代码阅读的更加容易，为获得每个纹理坐标的正确偏移值。

```
#define  WOOD_TC_OFFSET         0
#define  BRICK_TC_OFFSET        8
#define  FLOOR_TC_OFFSET        16
#define  CEILING_TC_OFFSET      24
```

绘制隧道

完整的隧道是这样的:



需要注意的第一件事是我们只使用了一个对象。绘制隧道的代码没有创建更多的对象，我们只使用了单独一个，只是我们操纵它，让它看起来像多重的。事实上，左边墙有5个，右边有5个，上下各有10个。

所以我猜，我可以说本教程介绍了最全的代码重用。

在我们绘制任何事情前，我们需要设置 OpenGL 绘制纹理映射的状态。下面的代码，你应该非常的熟悉了。

```
glBindTexture(GL_TEXTURE_2D, textures[0]);
glVertexPointer(3, GL_FLOAT, 0, elementVerticies);
glEnableClientState(GL_VERTEX_ARRAY);
glEnable(GL_TEXTURE_2D);
```

```
glEnableClientState(GL_TEXTURE_COORD_ARRAY);
```

我们要告诉 OpenGL 我们的顶点数组，告诉它去使用它。改变为纹理映射并且告诉它去使用它。我们不能告诉 OpenGL 关于纹理坐标数组，因为我们将取决于构成的矩形，我们正在绘制的。

首先，我们绘制地板。这个地板由两块我们之前定义的矩形组成。移动到屏幕的下半部分，旋转90度，让它看起来像个地板。然后并排的绘制。这个代码如下：

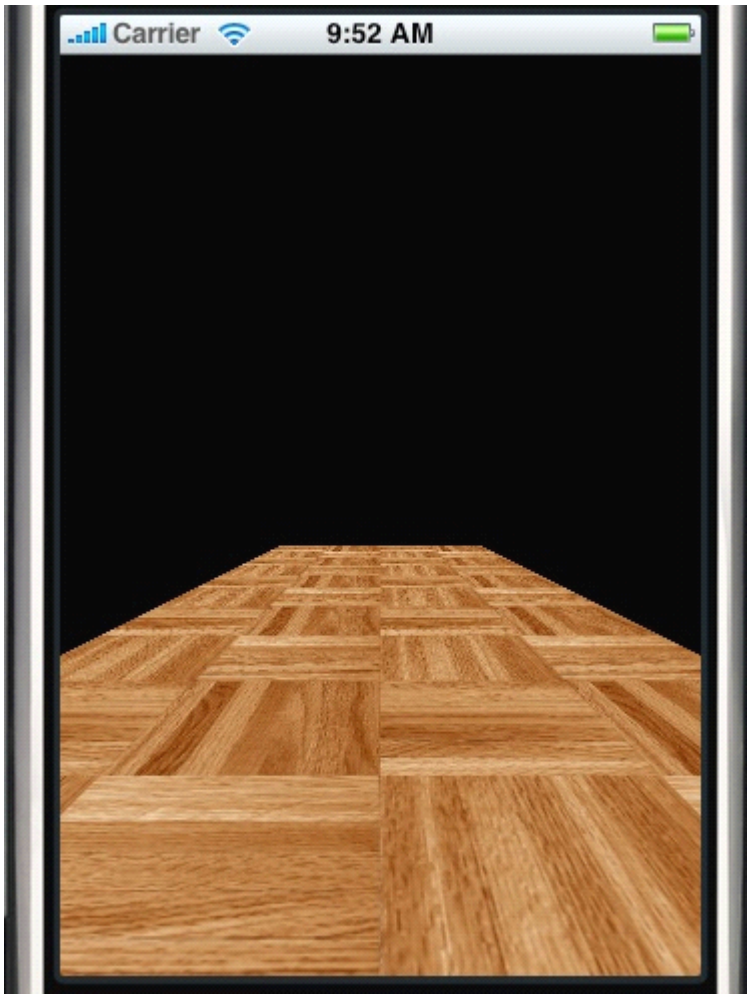
```
// Draw the Floor
// First, point the texture co-ordinate engine at the right offset
glTexCoordPointer(2, GL_FLOAT, 0, &combinedTextureCoordinate[FLOOR_TC_OFFSET]);
for (int i = 0; i < 5; i++) {
    glPushMatrix();
    {
        glTranslatef(-1.0, -1.0, -2.0+(i*-2.0));
        glRotatef(-90.0, 1.0, 0.0, 0.0);
        glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    }
    glPopMatrix();

    glPushMatrix();
    {
        glTranslatef(1.0, -1.0, -2.0+(i*-2.0));
        glRotatef(-90.0, 1.0, 0.0, 0.0);
        glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    }
    glPopMatrix();
}
```

我们用一个 for 循环来循环5次。每次循环中，都有两个地板的矩形被绘制出来。这里需要重视的改变是调用了 glTexCoordPointer():

```
glTexCoordPointer(2, GL_FLOAT, 0,
                                  &combinedTextureCoordinate[FLOOR_TC_OFFSET]);
```

数组的参数指针需要通过开始坐标的正确地址。看下开始定义的宏。

点击 “Build & Go”:

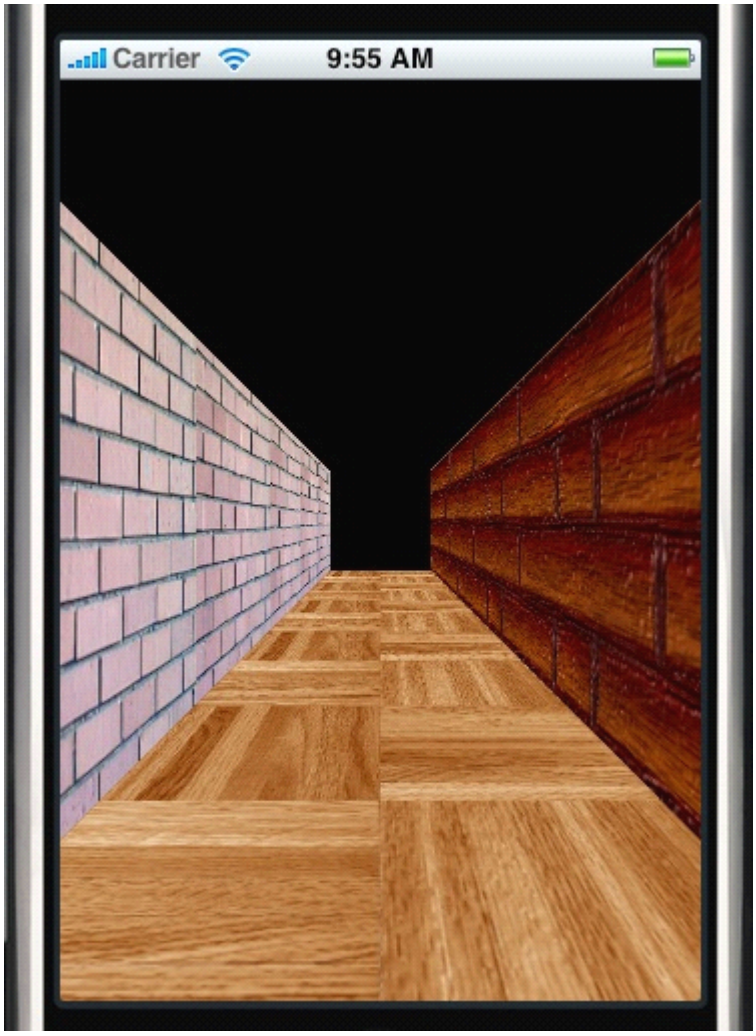


现在，轮到墙面了。过程是相同的，但是这一次，我们把它们在 x 轴上移动并且在 y 轴上旋转。

```
// Draw the walls
// This time we'll change the texture coordinate array during the drawing
for (int i = 0; i < 5; i++) {
```

```
            glPushMatrix();
            {
                glTexCoordPointer(2, GL_FLOAT, 0, &combinedTextureCoordinate[BRICK_TC
_OFFSET]);
                glTranslatef(-1.0, 0.0, -2.0+(i*-2.0));
                glRotatef(-90.0, 0.0, 1.0, 0.0);
                glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
            }
            glPopMatrix();

            glPushMatrix();
            {
                glTexCoordPointer(2, GL_FLOAT, 0, &combinedTextureCoordinate[WOOD_TC
_OFFSET]);
                glTranslatef(1.0, 0.0, -2.0+(i*-2.0));
                glRotatef(-90.0, 0.0, 1.0, 0.0);
                glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
            }
            glPopMatrix();
        }
```

请注意，我已经修改了纹理坐标的偏移值。因为我们要使用不用的纹理去绘制左边及右边。

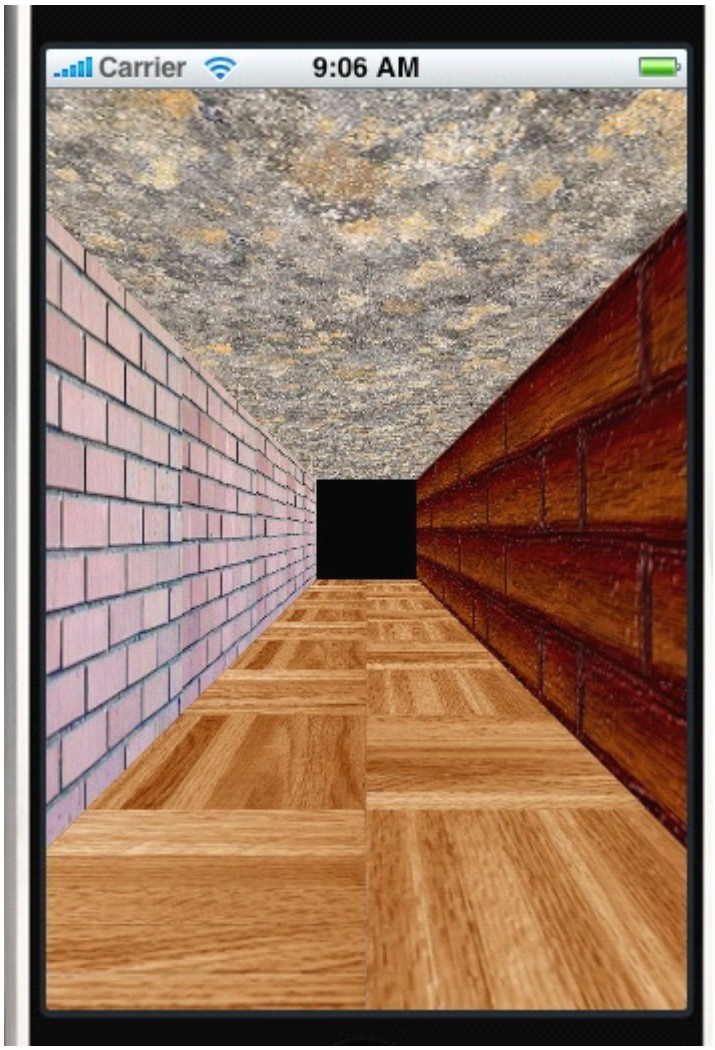


最后，我们绘制天空。和地板相同，只是增加 y 轴而不是减少。

```
// Draw the ceiling
// Start by setting the texture coordinate pointer
glTexCoordPointer(2, GL_FLOAT, 0, &combinedTextureCoordinate[CEILING_TC_OFFSET]);
for (int i = 0; i < 5; i++) {
    glPushMatrix();
    {
        glTranslatef(-1.0, 1.0, -2.0+(i*-2.0));
        glRotatef(90.0, 1.0, 0.0, 0.0);
        glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
    }
    glPopMatrix();
    glPushMatrix();
```

```
        {
            glTranslatef(1.0, 1.0, -2.0+(i*-2.0));
            glRotatef(90.0, 1.0, 0.0, 0.0);
            glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
        }
        glPopMatrix();
    }
```

这就是我们完成的隧道.



当然，这个没有什么特别的。这只是为给我们一个合适的场景来表示各种绘制效果。在这里我们开始处理，比如屏幕管理，从世界底图里渲染地板和墙壁。

纹理渲染：轻松

纹理渲染的过程就是由已经被绘制到屏幕上的来创建一个新的纹理。所以不是从我们的应用程序包里加载纹理，我们让 OpenGL 渲染屏幕，这时候我们复制全部或部分的场景到我们的一个缓冲区，然后在提交它完成屏幕之前重新添加它。

哦！深呼吸。这不是很复杂，并且这很有用。

一个简单的例子，纹理渲染是复制你渲染缓冲中的一部分。你应该这样想，当我们渲染缓冲中的光线效果，透明度和混合都已经完成了。所以你只需要复制它就可以使用这些效果了，而不需要重新计算灯光，阴影等。

I wasn't going to cover render to texture yet as it's probably a bit further on than where I'm up to so far, but there's been a few requests so I will cover the easy way to achieve this effect now. We won't see the best yet because we have not yet covered details such as lighting but you'll get the idea.

在我们隧道的世界里，我们要添加一个图片在隧道的尽头，并且使用提供的纹理来支持反射效果。

首先一个事，我们添加一个对象。

这不是光; 在我隧道的尽头是 Romo

第一件事，我们加载两张新的纹理。一个是背景效果，然后我们绘制第二张纹理（Romo），我的英雄，在前面。这样我就可以展示混合的纹理效果了。

增加两行新的函数在 initWithCoder[]:

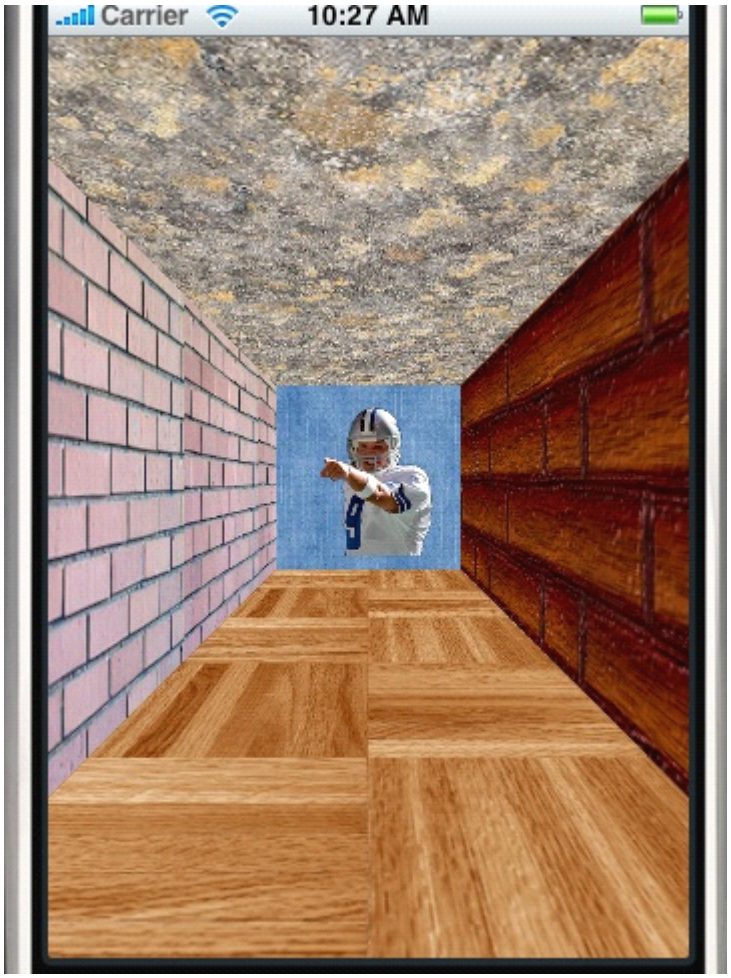
```
[self loadTexture:@"bluetex.png" intoLocation:textures[1]];
[self loadTexture:@"romo.png" intoLocation:textures[2]];
```

在绘制天花板之后，加载我们的纹理，然后我们将产生隧道尽头。首先我们需要设置标准的纹理坐标数组从我们纹理中，然后我们绘制不透明的纹理矩形，然后是部分透明的矩形。

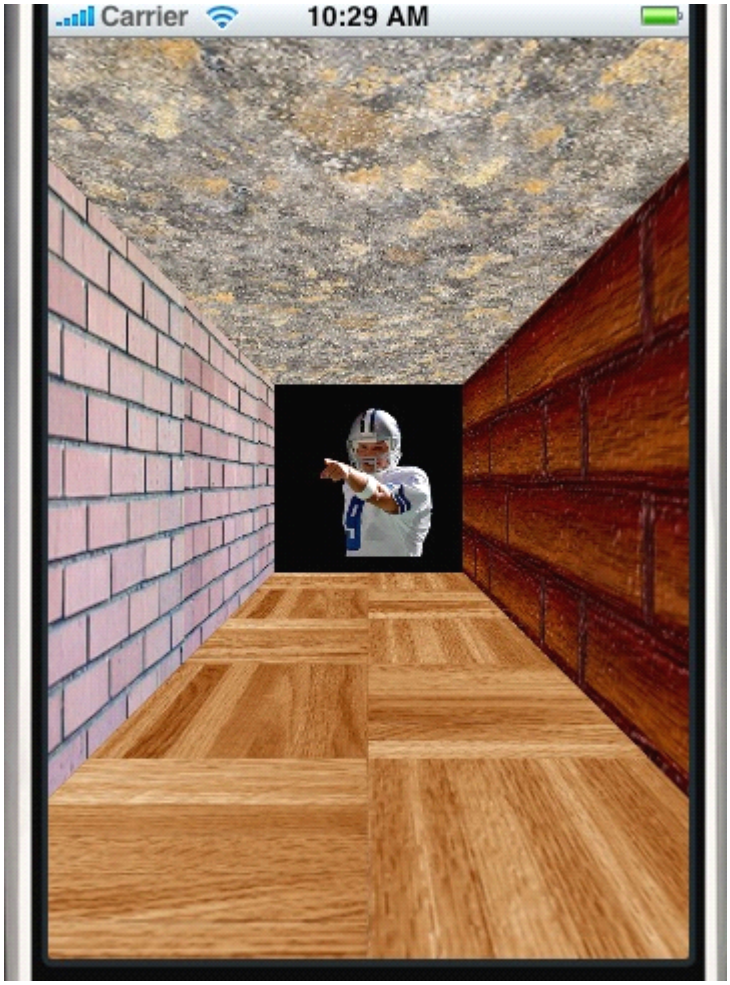
```
const GLfloat standardTextureCoordinates[] = {
    0.0, 1.0,
    0.0, 0.0,
    1.0, 0.0,
    1.0, 1.0
};
// Draw the Blue texture
glBindTexture(GL_TEXTURE_2D, textures[1]);
glTexCoordPointer(2, GL_FLOAT, 0, standardTextureCoordinates);
glPushMatrix();
{
    glTranslatef(0.0, 0.0, -6.0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
}
glPopMatrix();
glBindTexture(GL_TEXTURE_2D, textures[2]);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
glPushMatrix();
{
    glTranslatef(0.0, 0.0, -5.9);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
}
glPopMatrix();
glDisable(GL_BLEND);
```

有了对混合教程的失败例子来比较，这是一个好的多的例子。记不记得，在混合教程里面，你需要首先绘制不透明的物体，然后在绘制半透明的物体？这就是我们要做的。Romo 的图片只是其中一个，还有一个相同尺寸的蓝色纹理。

绘制结束以后就是这样:



通过这个混合的例子。注销掉 glEnable(GL_BLEND), 你只需要注销掉这行。你就可以得到以下的内容：



romo 的纹理图片掩盖了后面的蓝色纹理，注意到了没？这是非混合。我这里给你看到的是，仅仅是绘制一个纹理，包罗所有效果。

反注释 glEnable(GL_BLEND). 我们将使用的纹理渲染将不需要这个效果。现在我们设置。我将告诉你纹理渲染有那些过程。

纹理渲染：过程
纹理渲染的过程是很简单。只有四步：

1。创建渲染纹理是从那个从我们渲染图片复制出内容而创建的目标纹理。使用的方法完全同其他的纹理加载，你可以使用 loadTexture[].

2。渲染到你的屏幕。我们是这样做的;我们有 romo 在隧道里面的一部分混合效果以证明我们是从渲染缓冲中复制出来的。

3。从我们第一步设置的纹理中复制一部分（或全部）出来。

4。渲染新的纹理。

现在你知道四个步骤是什么了，只有第3步是全新的，其他的将有新的内容，但是你会非常熟悉。

创建纹理渲染

通常情况下，我们想用做纹理的图片，都是加载它，格式化它，然后调用 glTexImage2D() 发送它到 OpenGL。如果一个渲染纹理，我们步需要用了，但是我们依然需要获得 OpenGL 开启的空间，这是是从渲染屏幕中复制出的当前的存储纹理的空间。这是很简单的。到 init WithCoder[]，然后在加载纹理之后，为了渲染纹理添加下面的代码

```
// Render to Texture texture buffer setup
glBindTexture(GL_TEXTURE_2D, textures[3]);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 128, 128, 0, GL_RGBA,
                                                    GL_UNSIGNED_BYTE, nil);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
```

第一件我们要做的事，告诉 OpenGL 我们纹理在工作，并且选择下一个可用的纹理 id，然后，下面一行，我们创建了一个和我们加载文件一样的内存空间。

这里是两者的不同。首先，我们在纹理尺寸里写死了为128x128，你可以选择你喜欢的尺寸，但是必须是2的幂。

最后的参数我已经设置为 nil（或 NULL 如果您愿意） 。最后的参数，通常包含一个缓冲区包含我们的图象数据。由于我们没有图像数据，我们只是让 OpenGL 的知道，这将创建纹理的本身，而是没有填写任何数据到分配的内存。

最后，我们设置过滤参数。

渲染到屏幕上

这部分是自我解释。我们要做的就是写入数据到我们的纹理缓冲，然后到第3步。

复制部分纹理缓冲区域到我们的渲染纹理

有多种方式可以做到这点。在这个教程种我们只介绍一种最快的，但是支持了所有的功能。

我们要做的就是复制我们渲染缓冲中的一部分，同时包含了所有效果，如混合，或者其他我们做的效果。

在完成屏幕绘制的函数之后，我们增加下面几行。

```
glBindTexture(GL_TEXTURE_2D, textures[3]);
glCopyTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, 100, 150, 128, 128);
```

不管你相不相信，你已经做好了纹理渲染。第一行我们都知道，我们告诉了 OpenGL 使用那个纹理。调用 glCopyTexSubImage2D()是一个复杂的工作。这个函数获得了当前渲染缓冲中，复制其中的一部分并且存储到当前激活纹理中。

我们复制了（100，150）这点上，尺寸为128x128的像素到我们的纹理。这些参数如下：

glCopyTexSubImage2D(

target	在 OpenGL ES 一般是 GL_TEXTURE_2D
level	详细级别，将包括投影图

xoffset &yoffset 这是生成纹理目标的 x,y 的偏移值。

x & y 这是源（纹理缓冲）的 x,y 偏移值。

width & height 将复制的图片尺寸

);

需要注意的是，你是从2d 空间里复制出的数据，不包含 z 坐标，所以这里是不支持3d 空间的坐标的。你需要重新设定视窗空间。

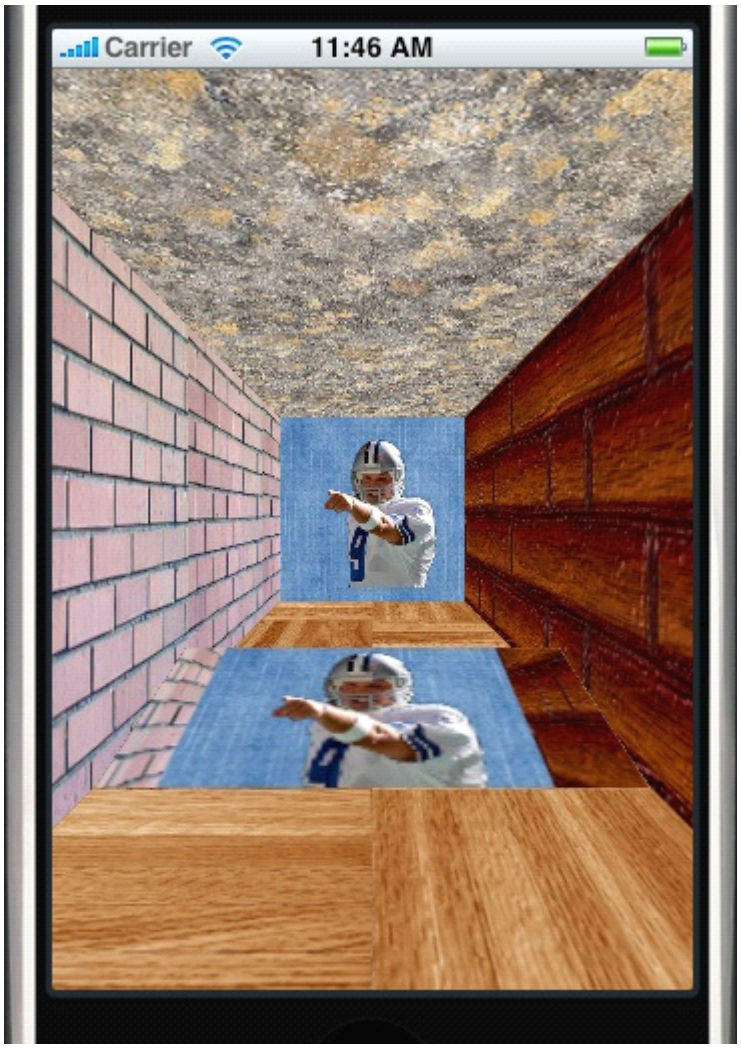
好了，纹理已经从渲染空间成功的创建了，现在我们需要使用它。

渲染我们新的纹理

最后部分的代码如下：

```
glPushMatrix();
{
    glTranslatef(0.0, -1.0, -2.0);
    glRotatef(-75.0, 1.0, 0.0, 0.0);
    glDrawArrays(GL_TRIANGLE_FAN, 0, 4);
}
glPopMatrix();
```

我们移动了一些，并旋转了下，所以就如下了：



西蒙 iphone-OpenGL ES 教程-13 在 3D 中移动

OpenGL ES 13 – 在 3D 中移动

（译者：那个。其实，实际上，我又跳了一章，12节，如何判断在2d 世界里点中一个物体，因为我觉得我们这大部分的游戏开发者都已经使用了花生大大的代码，也比较容易的在2d 世界里开发游戏了，这个章节对我们的用处不大。所以我又偷懒了。）

禽流感，忙碌，母亲节，和其他一些东西，这就意味着，我不在此博客一段时间。所以，现在是时候做事情了。

3D 中的移动

我的时间很紧迫，所以今天会是一个真正的快速入门。我们将开始建立一个“真正的”3d 世界，从地板开始建立一切的东西。然而，在我开始之前，我像先介绍如何在3d 中移动。

今天，我们将开始一些新的代码，围绕着地板，进行触摸及移动。在使用触摸，我们可以转左，转右，前进及后退。没有跑，没有抬头看，仅仅是走，因为这更加的容易。我不要实现他们是为了没有 iPod 及 iPhone 的朋友也可以在模拟器上实现它。

在开始之前，你先下载下面的开始函数：

OpenGL ES 13 - Starter

这里没有太多的代码，只是让我们知道我们要做什么。

虚拟的照相机

我想我跟你提过了，人们用照相机在3d 世界中观察事物，虽然 OpenGL 中实际上不存在照相机的。所以你在屏幕中移动的时候，你实际在移动所有的对象，运动的感觉的产生不是因为创建了一个照相机来看电影，而是因为将世界中的所有对象以（0，0，0）为相对坐标移动。

听上去好像这样的工作量非常的大，其实没有。因为根据你的应用，会执行很多的优化工作，即使很大的世界也没有太多的工作量。我以后会提到它。

为了工作更方便，我为本教程带来了帮手，来自于 OpenGL ES 的大哥 GL U 的 library 中的函数：gluLookAt()。

通常我不提及 OpenGL 的优点，但我想大部分的人都知道什么是 GLU library。遗憾的是 OpenGL ES 不支持这个函数库，这意味了我们少了很多有用的功能。移植这些功能不需要移植整个 library，只需要实现少许我们需要的功能。

我找到了 gluLookAt()的源代码，所以这些的代码的版权不是我的。

使用 gluLookAt()

这个函数是如此的简单使用，以至于你只要了解它就能使用，让我们来看看原型：

```
void  gluLookAt(  GLfloat  eyex,
                  GLfloat  eyey,
                  GLfloat  eyez,
                  GLfloat  centerx,
                  GLfloat  centery,
                  GLfloat  centerz,
                  GLfloat  upx,
                  GLfloat  upy,
                  GLfloat  upz)
```

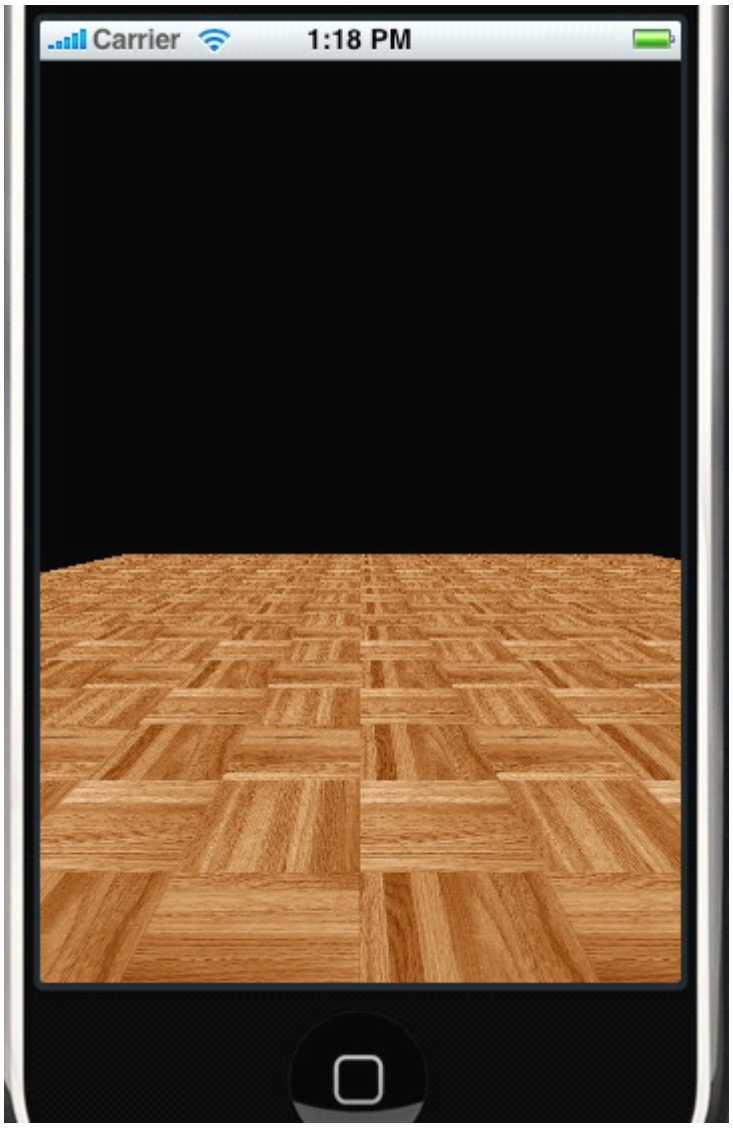
我知道9个参数看起来有点令人生畏，但你可以分解。前3个是指眼睛的位置，在这里你要找的，只是 x,y,z 的坐标。

这第2组3个参数是指你想看哪里，又一个 x,y,z 的坐标。

最后，我们看最后3个“up”的向量。前2个坐标就是我想要的效果。

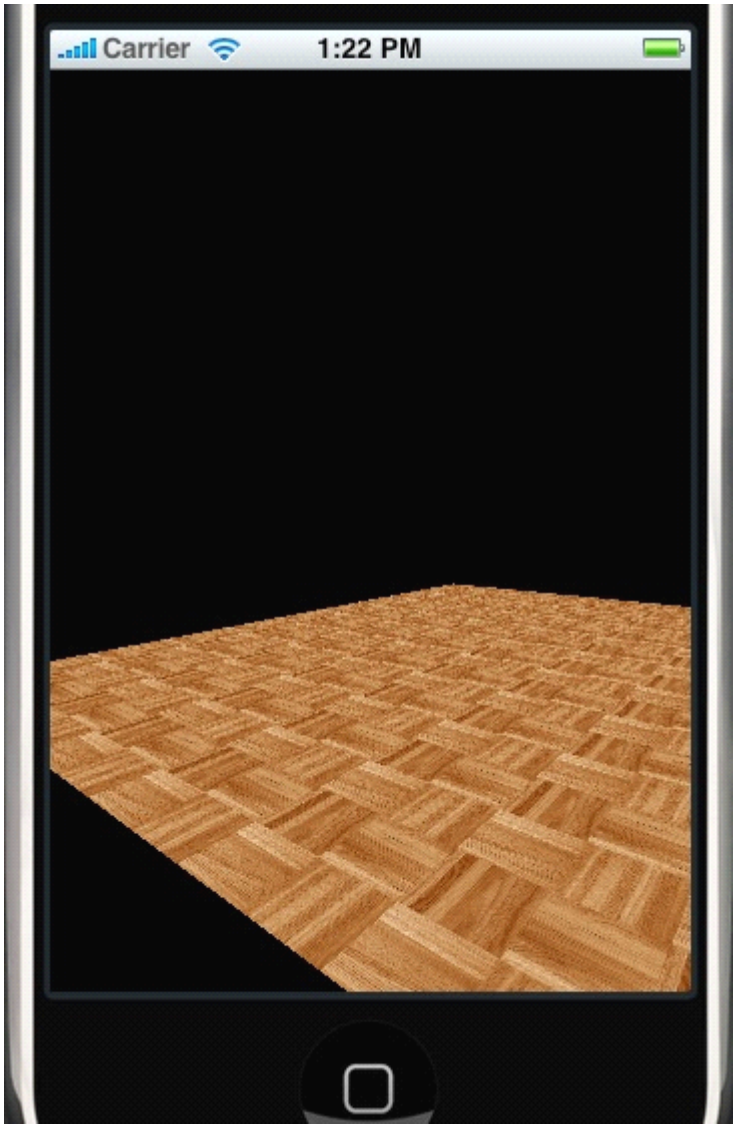
所以，眼睛的坐标就是神秘的照相机的位置，它们在你的世界坐标里。从这里看你的世界。那个“中心”坐标就是你面向那里。如果中心的 y 坐标高于你眼睛的 y 坐标，那么你就朝上看，如果低于呢，你就朝下看。

所以，在我们开始的工程里，我们就已经准备号设置如何移动了。现在我们看看绘制好的地板。



首先让我们使用下 glLookAt()。到 drawView，添加下面函数：

```
glLoadIdentity();
gluLookAt(5.0, 1.5, 2.0,          // Eye location, look “from”
          -5.0, 1.5, -10.0,       // Target location, look “to”
          0.0, 1.0, 0.0);         // Ignore for now
```



单一的功能，我们从一个地方，看到另一个地方，改变参数，看看会发生什么。

3D 中的运动

现在你又把握处理 gluLookAt()，让我们用它来模拟在地板上行走，我们只能移动(X,Z),也就是说，不要移动 y，就是高度。

所以，思路会到 gluLookAt()，想想，在3d 世界中移动，你需要什么信息？

你需要:

你观察的位置或者说眼

你面向的位置或者说“中心”

一旦你知道了这两项，你就要准备好用户输入，以让用户控制移动。

设置以下变量来改变眼睛及中心

```
GLfloat eye[3];// Where we are viewing from
GLfloat center[3];// Where we are looking towards
```

回到 initWithCoder: 在调用 gluLookAt()之前，初始化下：

```
eye[0] = 5.0;
eye[1] = 1.5;
eye[2] = 2.0;

center[0] = -5.0;
center[1] = 1.5;
center[2] = -10.0;
```

到 drawView: 调用 gluLookAt():

```
gluLookAt(eye[0], eye[1], eye[2], center[0], center[1], center[2],
          0.0, 1.0, 0.0);
```

为运动做的准备

在我们开始执行触摸事件并在世界中移动之前，我们需要在头文件中设置一些东西。来到头文件，我们需要创建一个新的枚举类型。

首先，设置我们的步行速度及转身速度：

```
#define WALK_SPEED 0.005
#define TURN_SPEED 0.01
```

下一步，我们创建一个枚举，来表示我们正在做什么，如下：

```
typedef enum __MOVMENT_TYPE {
MTNone = 0,
MTWalkForward,
MTWAlkBackward,
MTTurnLeft,
MTTurnRight
} MovementType;
```

所以，在我们运行 app 的时候，我们可以一直是站着，或者前面，或者后退，或者转左，或者转右。

最后，我们定义个变量保存但前的运动状态：

```
MovementType currentMovement;
```

别忘了到 initWithCoder: 函数里面去设置 currentMovement 的默认值:

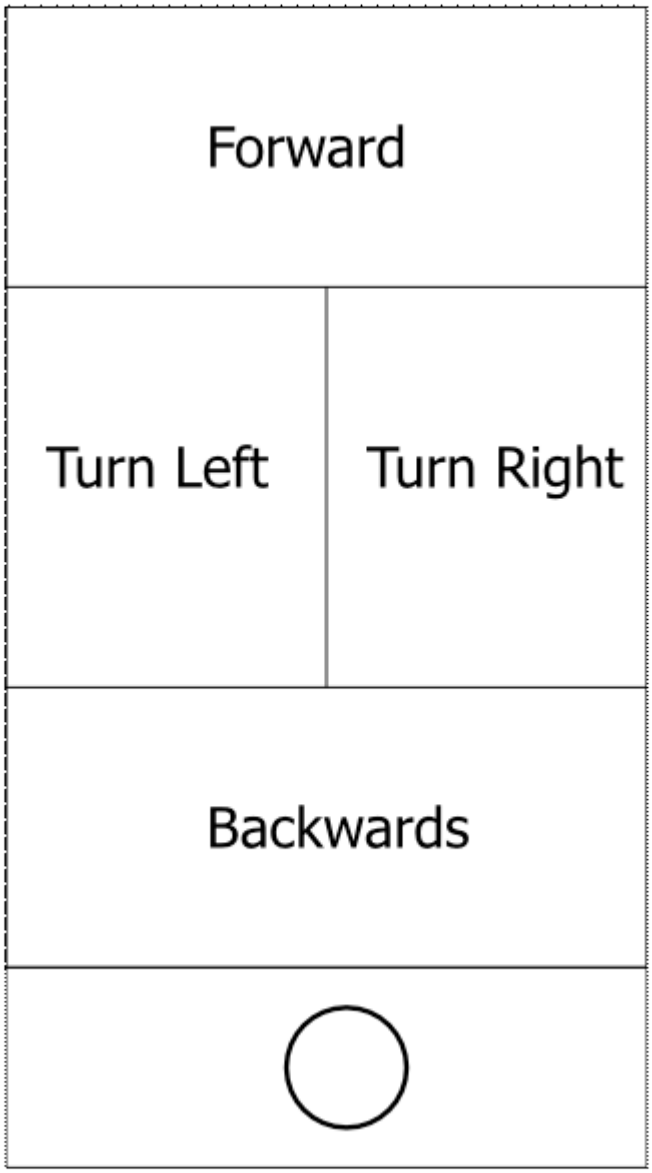
```
currentMovement = MTNone;
```

获得触摸

好的，我们已经进行了基本的介绍，让我们开始实际的处理工作。如果你还记得我之前的教程，你应该知道触摸状态四种，今天我们使用

touchesBegan 和 touchesEnded.

为了确定行动的种类，我们把屏幕分为四部分。



开始进行触摸的函数：

```
- (void)touchesBegan:(NSSet *)touches withEvent:(UIEvent *)event {

    UITouch *t = [[touches allObjects] objectAtIndex:0];
    CGPoint touchPos = [t locationInView:t.view];

    // Determine the location on the screen. We are interested in iPhone
    // Screen co-ordinates only, not the world co-ordinates
```

```
    //  because we are just trying to handle movement.
    //
    // (0, 0)
    //  +-----------+
    //  |           |
    //  |    160    |
    //  |-----------| 160
    //  |     |     |
    //  |     |     |
    //  |-----------| 320
    //  |           |
    //  |           |
    //  +-----------+ (320, 480)
    //

    if (touchPos.y < 160) {
        // We are moving forward
        currentMovement = MTWalkForward;

    } else if (touchPos.y > 320) {
        // We are moving backward
        currentMovement = MTWAlkBackward;

    } else if (touchPos.x < 160) {
        // Turn left
        currentMovement = MTTurnLeft;
    } else {
        // Turn Right
        currentMovement = MTTurnRight;
    }
}
```

在玩家点击屏幕的任意位置后，就会将运动状态设置为指定的状态，同时，在松手时释放这样的状态：

```
- (void)touchesEnded:(NSSet *)touches withEvent:(UIEvent *)event {

    currentMovement = MTNone;
}
```

这些时方法的实现。最后，我们需要一个来处理这些触摸事件，这一次，我们需要添加一个方法声明的接口，回到头文件，添加下面的方法：

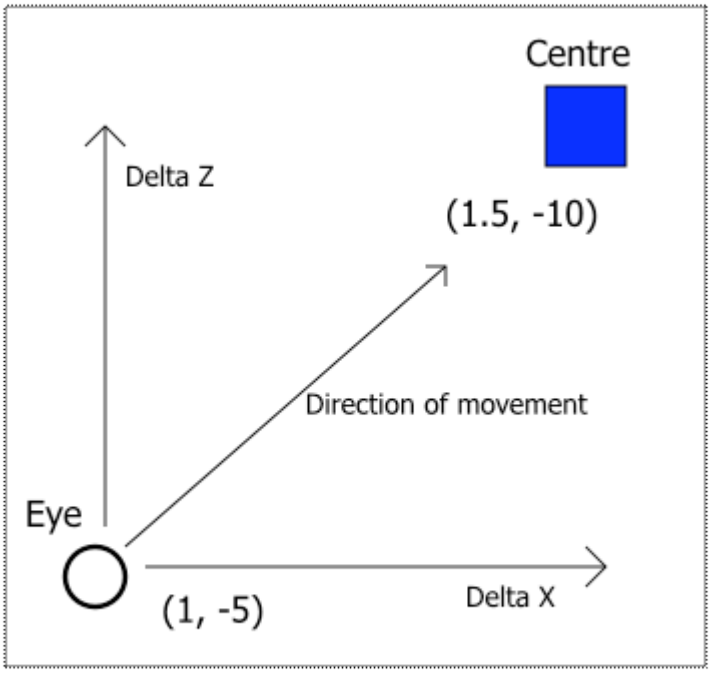
```
- (void)handleTouches;
```

然后，切换回来，实现这个方法，就是这个方法，我们要通过计算来执行我们运动的3d 世界。

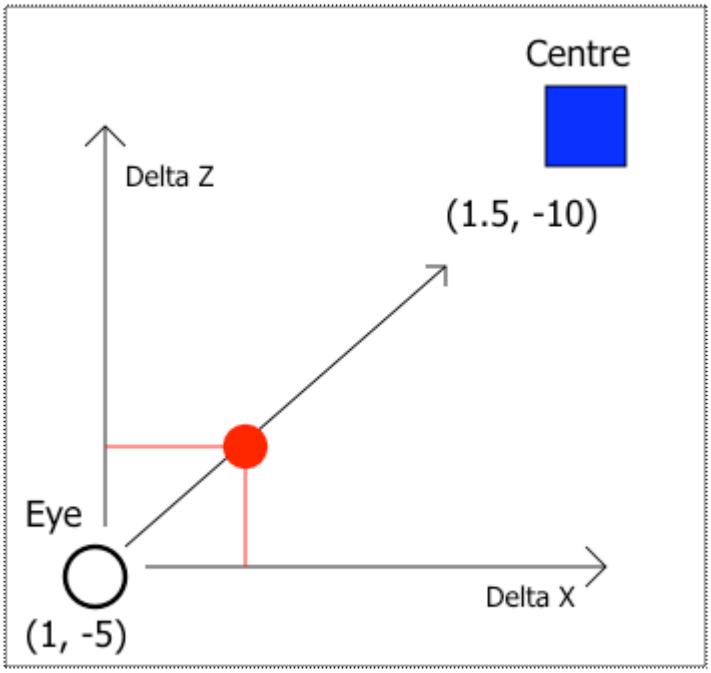
3D 空间的移动理论

让我们来看看第一次走路。当用户告诉我们向前走，我们需要了解的不仅是我们朝向的位置，同时还要了解我们的目标位置。这个朝向位置就是我们的当前位置，而看到的目标就是我们的目标位置，我们需要从当前位置向目标位置移动。

一图胜千言，下面的图片很好的说明了我们的当前位置和我们预定的目标位置。



在运动函数里面，我们知道这两点之间，x 及 z 坐标的距离。我们要做的就是，将当前坐标乘以”速度”获得新的 x,z 坐标。像这样：



我们可以很容易的得到红色点的坐标.

我们先设计目标位置的相对坐标 deltaX 和 deltaZ:

deltaX = 1.5 - 1.0 = 0.5

deltaZ = -10 - (- 5.0) = -5.0

所以现在乘上我们的步行速度：

xDisplacement = deltaX * WALK_SPEED

= 0.5 * 0.01

= 0.005

zDisplacement = deltaZ * WALK_SPEED

= -5.0 * 0.01

= 0.05

因此，那个红色的新点位置就是:

eyeC + CDisplacement

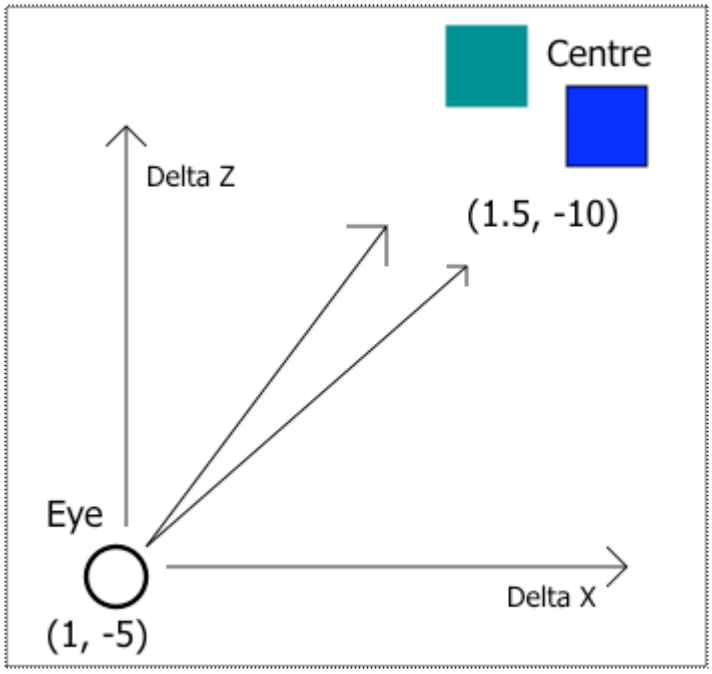
(eyex + xDisplacement, eyey, eyez + zDisplacement)

= (0.005+1.0, eyey,(-10)+ 0.05)

= (1.005, eyey, -9.95)

现在，这种方法不是没有缺点。主要的问题是，如果目标与当前位置距离越远，那么“步行速度”也就越快。

现在，让我们看下转左和转右。



因为我们在模拟旋转。实际上目标位置就是在一个圆上移动。

这里的关键在于，你要明白我说的是什么。你可以把当前所在位置想像为圆心，目标位置是圆边。转向就是在圆边上移动。

换句话说，这就是:

```
newX = eyeX + radius * cos(TURN_SPEED)*deltaX -
                    sin(TURN_SPEED)*deltaZ

newZ = eyeZ + radius * sin(TURN_SPEED)* deltaX +
                    cos(TURN_SPEED)*deltaZ
```

处理触摸及涉及到运动

现在，让我们把这个转化为实践。

首先，实现开始的方法和一些基本知识：

```
- (void)handleTouches {

    if (currentMovement == MTNone) {
        // We're going nowhere, nothing to do here
        return;
    }
```

首先，看看我们是不是在移动，如果没有操作，就什么事情也不做。

然后，无论我们是在移动还是转弯，我们都需要知道相对距离 deltaX 和 deltaZ 的值。我找一个临时数组把我们保存起来。

```
    GLfloat vector[3];

    vector[0] = center[0] - eye[0];
    vector[1] = center[1] - eye[1];
    vector[2] = center[2] - eye[2];
```

我没有计算 y 的相对值，因为是没有意义的。

现在我们用一个 switch 来判断当前状态并进行处理

```
    switch (currentMovement) {
        case MTWalkForward:
            eye[0] += vector[0] * WALK_SPEED;
            eye[2] += vector[2] * WALK_SPEED;
            center[0] += vector[0] * WALK_SPEED;
            center[2] += vector[2] * WALK_SPEED;
            break;

        case MTWAlkBackward:
            eye[0] -= vector[0] * WALK_SPEED;
            eye[2] -= vector[2] * WALK_SPEED;
            center[0] -= vector[0] * WALK_SPEED;
            center[2] -= vector[2] * WALK_SPEED;
            break;

        case MTTurnLeft:
            center[0] = eye[0] + cos(-TURN_SPEED)*vector[0] -
sin(-TURN_SPEED)*vector[2];
            center[2] = eye[2] + sin(-TURN_SPEED)*vector[0] +
cos(-TURN_SPEED)*vector[2];
            break;

        case MTTurnRight:
            center[0] = eye[0] + cos(TURN_SPEED)*vector[0] - sin(TURN_SPEED)*vect
or[2];
            center[2] = eye[2] + sin(TURN_SPEED)*vector[0] + cos(TURN_SPEED)*vec
```

```
tor[2];
                    break;
        }
}
```

这里实现的算法就是我刚才说的。

使这一切关联起来

回到 drawView 函数，并在 gluLookAt()函数下面添加这行:

```
    [self handleTouches];
```

And that's it, we're done!

Hit "Build and Go" now! Right now!!