[转载]从零开始学习OpenGL ES之四 - 光效

2011-9-14 23:36 | 发布者: Iven | 查看: 11067 | 评论: 9

摘要: 图形图像, 编程,编程, OpenGL ES, 教程,OpenGL ES 3D

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：

相关分类

[iTyran原 [翻译]OpenGL ES
[子龙山人翻 从零开始学习

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：

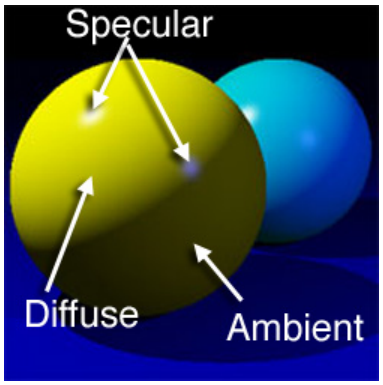
```
glEnable(GL_LIGHT0);
```

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入注目，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在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。

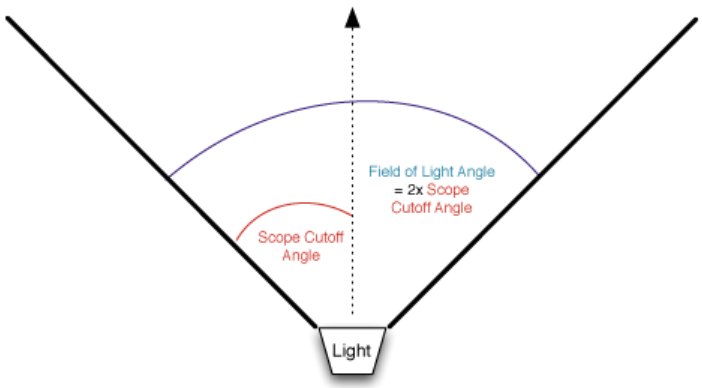
每个人都知道两点可以定义一条线段，那么空间中的一点怎么可能指定方向？这是因为存在一个隐性的第二点作为起点，即原点。如果你从原点画一条线到向量定义的点，那就是向量代表的方向。向量还可用于表示速度和距离，一个远离原点的点表示速度越快或距离更远。在大部分的OpenGL应用中，并未使用距离原点的距离。实际上，在大部分使用向量的情况下，我们需要将向量标准化（normalize） 为长度 1.0。关于向量的标准化，随着我们继续深入将会讨论到。现在你只需知道，如果你希望定义一个定向光，那么你 必须创建一个定义了光的方向的向量。下例演示了定义一个沿z轴而下的光源：

```
    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);
```

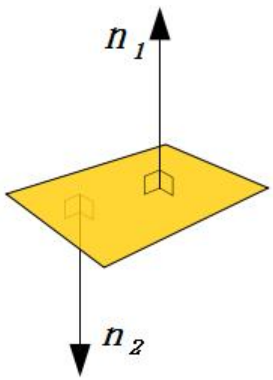
```
    // Turn the first light on
    glEnable(GL_LIGHT0);

    // 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 ax
is
    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。

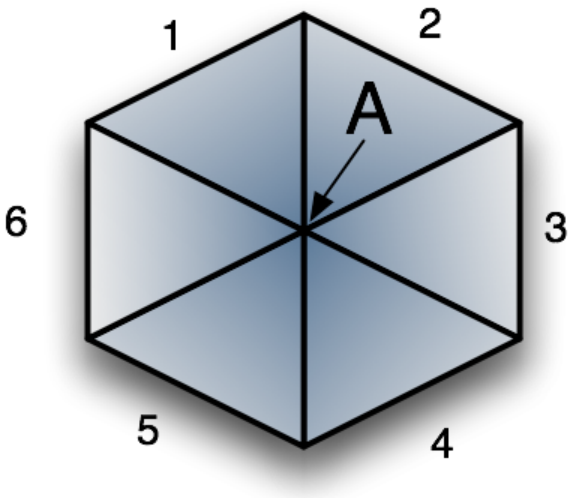
OpenGL 要求我们为各使用的顶点提供法线。计算一个三角形的表面法线是很简单的，它是三角形两边的叉积。代码如下：

```
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, norm
als[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：

如果你不想输入所有代码，可下载源代码。

19

鲜花 握手 雷人 路过 鸡蛋

刚表态过的朋友 (19 人)

々枫々 45553283... 我本俗人 bullet. wzx11011... 怀旧 xiupoman yuxiang11... iam_gaowei tb706

yjh4866 飞翔 ccboby sen 红沙尘 fspinach 286820549 fox qq963922...

邀请 收藏

最新评论 发表评论

coolbabylch 2012-11-13 18:47 引用

好东西呀。

tb706 2012-10-10 10:29 引用

好东西呀。

功夫杨 2012-7-3 14:12 引用

顶顶顶

myhapi 2012-2-20 11:28 引用

正在学习这个，受教啦

mark006002 2012-2-3 10:13 引用

内容真豐富！

cl890305 2012-1-20 13:40 引用

每天学一点，顶一个

panzhiccp 2011-12-28 10:37 引用

我顶顶顶