基于MFC的OpenGL绘图

一、简介

GDI是通过设备句柄（Device Context以下简称"DC"）来绘图，而OpenGL则需要绘制环境（Rendering Context，以下简称"RC"）。每一个GDI命令需要传给它一个DC，但与GDI不同，OpenGL使用当前绘制环境(RC)。一旦在一个线程中指定 了一个当前RC，在此线程中其后所有的OpenGL命令都使用相同的当前RC。虽然在单一窗口中可以使用多个RC，但在单一线程中只有一个当前RC。下面我将首先产生一个OpenGL RC并使之成为当前RC，这将分为三个步骤：设置窗口像素格式；产生RC；设置为当前RC。

二、MFC中的OpenGL基本框架

1、首先创建工程

用AppWizard产生一个MFC EXE项目，其他默认即可。

2、将此工程所需的OpenGL文件和库加入到工程中

在工程菜单中，选择"Build"下的"Settings"项。单击"Link"标签，选择"General"目录，在Object/Library Modules的编辑框中输入"opengl32.lib glu32.lib glut.lib glaux.lib"（注意，输入双引号中的内容，各个库用空格分开；否则会出现链接错误），选择"OK"结束。然后打开文件"stdafx.h"，加入下列头文件：

```
  #include <gl\gl.h>
#include <gl\glu.h>
```

3、改写OnPreCreate函数并给视图类添加成员函数和成员变量

OpenGL需要窗口加上WS_CLIPCHILDREN（创建父窗口使用的Windows风格，用于重绘时裁剪子窗口所覆盖的区域）和WS_CLIPSIBLINGS（创建子窗口使用的Windows风格，用于重绘时剪裁其他子窗口所覆盖的区域）风格。把OnPreCreate改写成如 下所示：

```
BOOL COpenGLDemoView::PreCreateWindow(CREATESTRUCT& cs)
{
   // TODO: Modify the Window class or styles here by modifying
   //  the CREATESTRUCT cs
   cs.style |= (WS_CLIPCHILDREN | WS_CLIPSIBLINGS);
   return CView::PreCreateWindow(cs);
}
```

产生一个RC的第一步是定义窗口的像素格式。像素格式决定窗口着所显示的图形在内存中是如何表示的。由像素格式控制的参数包括：颜色深度、缓冲模式和所支持的绘画接口。在下面将有对这些参数的设置。我们先在COpenGLDemoView的类中添加一个保护型的成员函数BOOL SetWindowPixelFormat(HDC hDC)（用鼠标右键添加）和保护型的成员变量：int m_GLPixelIndex;并编辑其中的代码如下：

```
BOOL COpenGLDemoView::SetWindowPixelFormat(HDC hDC)
{//定义窗口的像素格式
   PIXELFORMATDESCRIPTOR pixelDesc=
   {
      sizeof(PIXELFORMATDESCRIPTOR),
      1,
      PFD_DRAW_TO_WINDOW|PFD_SUPPORT_OPENGL|
      PFD_DOUBLEBUFFER|PFD_SUPPORT_GDI,
      PFD_TYPE_RGBA,
      24,
      0,0,0,0,0,0,
      0,
      0,
      0,
      0,0,0,0,
      32,
```

```
    0,
    0,
    PFD_MAIN_PLANE,
    0,
    0,0,0
  };

  this->m_GLPixelIndex = ChoosePixelFormat(hDC,&pixelDesc);
  if(this->m_GLPixelIndex==0)
  {
    this->m_GLPixelIndex = 1;
    if(DescribePixelFormat(hDC,this-
>m_GLPixelIndex,sizeof(PIXELFORMATDESCRIPTOR),&pixelDesc)==0)
    {
      return FALSE;
    }
  }

  if(SetPixelFormat(hDC,this->m_GLPixelIndex,&pixelDesc)==FALSE)
  {
    return FALSE;
  }
  return TRUE;
```

4、用ClassWizard添加WM_CREATE的消息处理函数OnCreate

至此，OpenGL工程的基本框架就建好了。但如果你现在运行此工程，则它与一般的MFC程序看起来没有什么两样。

5、代码解释

现在我们可以看一看Describe-PixelFormat提供有哪几种像素格式，并对代码进行一些解释：

PIXELFORMATDESCRIPTOR包括了定义像素格式的全部信息。

DWFlags定义了与像素格式兼容的设备和接口。

通常的OpenGL发行版本并不包括所有的标志(flag)。wFlags能接收以下标志：

PFD_DRAW_TO_WINDOW 使之能在窗口或者其他设备窗口画图；

PFD_DRAW_TO_BITMAP 使之能在内存中的位图画图；

PFD_SUPPORT_GDI 使之能调用GDI函数（注：如果指定了PFD_DOUBLEBUFFER，这个选项将无效）；

PFD_SUPPORT_OpenGL 使之能调用OpenGL函数；

PFD_GENERIC_FORMAT 假如这种象素格式由Windows GDI函数库或由第三方硬件设备驱动程序支持，则需指定这一项；

PFD_NEED_PALETTE 告诉缓冲区是否需要调色板，本程序假设颜色是使用24或 32位色，并且不会覆盖调色板；

PFD_NEED_SYSTEM_PALETTE 这个标志指明缓冲区是否把系统调色板当作它自身调色板的一部分；

PFD_DOUBLEBUFFER 指明使用了双缓冲区（注：GDI不能在使用了双缓冲区的窗口中画图）；

PFD_STEREO 指明左、右缓冲区是否按立体图像来组织。

PixelType定义显示颜色的方法。PFD_TYPE_RGBA意味着每一位(bit)组代表着红、绿、蓝各分量的值。PFD_TYPE_COLORINDEX 意味着每一位组代表着在彩色查找表中的索引值。本例都是采用了PFD_TYPE_RGBA方式。

- cColorBits定义了指定一个颜色的位数。对RGBA来说，位数是在颜色中红、绿、蓝各分量所占的位数。对颜色的索引值来说，指的是表中的颜色数。
- cRedBits、cGreenBits、cBlue-Bits、cAlphaBits用来表明各相应分量所使用的位数。
- cRedShift、cGreenShift、cBlue-Shift、cAlphaShift用来表明各分量从颜色开始的偏移量所占的位数。

一旦初始化完我们的结构，我们就想知道与要求最相近的系统象素格式。我们可以这样做：

m_hGLPixelIndex = ChoosePixelFormat(hDC, &pixelDesc);

ChoosePixelFormat接受两个参数：一个是hDc，另一个是一个指向PIXELFORMATDESCRIPTOR结构的指针& pixelDesc；该函数返回此像素格式的索引值。如果返回0则表示失败。假如函数失败，我们只是把索引值设为1并用 DescribePixelFormat得到像素格式描

述。假如你申请一个没得到支持的像素格式，则Choose-PixelFormat将会返回与你要 求的像素格式最接近的一个值。一旦我们得到一个像素格式的索引值和相应的描述，我们就可以调用SetPixelFormat设置像素格式，并且只需设置一 次。

现在像素格式已经设定，我们下一步工作是产生绘制环境(RC)并使之成为当前绘制环境。在COpenGLDemoView中加入一个保护型的成员函数BOOL CreateViewGLContext(HDC hDC)，并加入一个保护型的成员变量HGLRC m_hGLContext；HGLRC是一个指向rendering context的句柄。

```
BOOL COpenGLDemoView::CreateViewGLContext(HDC hDC)
{
   this->m_hGLContext = wglCreateContext(hDC);
   if(this->m_hGLContext==NULL)
   {//创建失败
      return FALSE;
   }

   if(wglMakeCurrent(hDC,this->m_hGLContext)==FALSE)
   {//选为当前RC失败
      return FALSE;
   }

   return TRUE;


}
```

在OnCreate函数中调用此函数：

```
int COpenGLDemoView::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
   if (CView::OnCreate(lpCreateStruct) == -1)
      return -1;

   // TODO: Add your specialized creation code here
   HWND hWnd = this->GetSafeHwnd();
   HDC hDC = ::GetDC(hWnd);
   if(this->SetWindowPixelFormat(hDC)==FALSE)
   {
      return 0;
   }
   if(this->CreateViewGLContext(hDC)==FALSE)
   {
      return 0;
   }
   return 0;
}
```

添加WM_DESTROY的消息处理函数Ondestroy( )，使之如下所示：

```
void COpenGLDemoView::OnDestroy()
{
   CView::OnDestroy();

   // TODO: Add your message handler code here
   if(wglGetCurrentContext()!=NULL)
   {
      wglMakeCurrent(NULL,NULL);
   }
   if(this->m_hGLContext!=NULL)
   {
      wglDeleteContext(this->m_hGLContext);
      this->m_hGLContext = NULL;
```

```
    }
}
```

最后，编辑COpenGLDemoView的构造函数，使之如下所示：

```
COpenGLDemoView::COpenGLDemoView()
{
    // TODO: add construction code here
    this->m_GLPixelIndex = 0;
    this->m_hGLContext = NULL;
}
```

至此，我们已经构造好了框架，使程序可以利用OpenGL进行画图了。你可能已经注意到了，我们在程序开头产生了一个RC，自始自终都使用它。这与大多数GDI程序不同。在GDI程序中，DC在需要时才产生，并且是画完立刻释放掉。实际上，RC也可以这样做；但要记住，产生一个RC需要很多处理器时 间。因此，要想获得高性能流畅的图像和图形，最好只产生RC一次，并始终用它，直到程序结束。

CreateViewGLContex产生RC并使之成为当前RC。WglCreateContext返回一个RC的句柄。在你调用 CreateViewGLContex之前，你必须用SetWindowPixelFormat(hDC)将与设备相关的像素格式设置好。 wglMakeCurrent将RC设置成当前RC。传入此函数的DC不一定就是你产生RC的那个DC，但二者的设备句柄(Device Context)和像素格式必须一致。假如你在调用wglMakeforCurrent之前已经有另外一个RC存在，wglMakeforCurrent 就会把旧的RC冲掉，并将新RC设置为当前RC。另外你可以用wglMakeCurrent(NULL, NULL)来消除当前RC。

我们要在OnDestroy中把绘制环境删除掉。但在删除RC之前，必须确定它不是当前句柄。我们是通过wglGetCurrentContext来了 解是否存在一个当前绘制环境的。假如存在，那么用wglMakeCurrent(NULL, NULL)来把它去掉。然后就可以通过wglDelete-Context来删除RC了。这时允许视类删除DC才是安全的。注：一般来说，使用的都是单线 程的程序，产生的RC就是线程当前的RC，不需要关注上述这一点。但如果使用的是多线程的程序，那我们就特别需要注意这一点了，否则会出现意想不到的后 果。

三、画图实例

下面给出一个简单的二维图形的例子（这个例子都是以上述框架为基础的）。

用Classwizard为COpenGLDemoView添加WMSIZE的消息处理函数OnSize，代码如下：

```
void COpenGLDemoView::OnSize(UINT nType, int cx, int cy)
{
    CView::OnSize(nType, cx, cy);

    // TODO: Add your message handler code here
    GLsizei width,height;
    GLdouble aspect;
    width = cx;
    height = cy;
    if(cy==0)
    {
        aspect = (GLdouble)width;
    }
    else
    {
        aspect = (GLdouble)width/(GLdouble)height;
    }
    glViewport(0,0,width,height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0,500.0*aspect,0.0,500.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

用Classwizard为COpenGLDemoView添加WM_PAINT的消息处理函数OnPaint，代码如下：

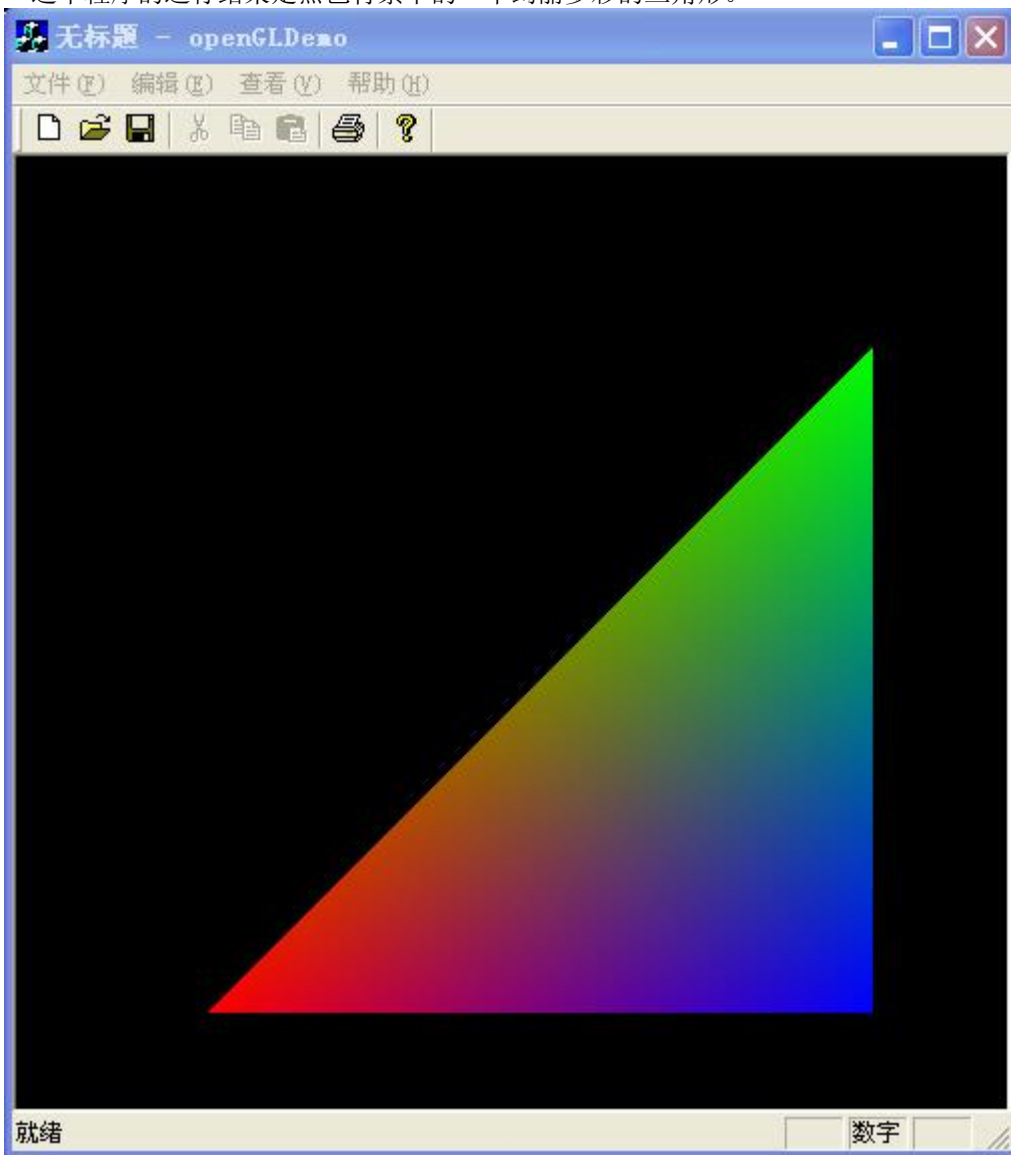
```
void COpenGLDemoView::OnPaint()
{
    CPaintDC dc(this); // device context for painting
```

```
    // TODO: Add your message handler code here

    // Do not call CView::OnPaint() for painting messages

    glLoadIdentity();
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POLYGON);
        glColor4f(1.0f,0.0f,0.0f,1.0f);
        glVertex2f(100.0f,50.0f);
        glColor4f(0.0f,1.0f,0.0f,1.0f);
        glVertex2f(450.0f,400.0f);
        glColor4f(0.0f,0.0f,1.0f,1.0f);
        glVertex2f(450.0f,50.0f);
    glEnd();
    glFlush();
}
```

这个程序的运行结果是黑色背景下的一个绚丽多彩的三角形。



这里你可以看到用OpenGL绘制图形非常容易，只需要几条简单的语句就能实现 强大的功能。如果你缩放窗口，三角形也会跟着缩放。这是因为OnSize通过glViewport(0, 0, width, height)定义了视口

和视口坐标。glViewport的第一、二个参数是视口左下角的像素坐标，第三、四个参数是视口的宽度和高度。

OnSize中的glMatrixMode是用来设置矩阵模式的，它有三个选项：GL_MODELVIEW、GL_PROJECTION、GL_TEXTURE。GL_MODELVIEW表示从实体坐标系转到人眼坐标系。GL_PROJECTION表示从人眼坐标系转到剪裁坐标系。GL_TEXTURE表示从定义纹理的坐标系到粘贴纹理的坐标系的变换。

glLoadIdentity初始化工程矩阵(project matrix)；gluOrtho2D把工程矩阵设置成显示一个二维直角显示区域。

这里我们有必要说一下OpenGL命令的命名原则。大多数OpenGL命令都是以"gl"开头的。也有一些是以"glu"开头的，它们来自OpenGL Utility。大多数"gl"命令在名字中定义了变量的类型并执行相应的操作。例如：glVertex2f就是定义了一个顶点，参数变量为两个浮点数， 分别代表这个顶点的x、y坐标。类似的还有glVertex2d、glVertex2f、glVertex3I、glVertex3s、 glVertex2sv、glVertex3dv……等函数。

那么，怎样画三角形 呢？我们首先调用glColor4f(1.0f, 0.0f, 0.0f, 1.0f)，把红、绿、蓝分量分别指定为1、0、0。然后我们用glVertex2f(100.0f, 50.0f)在(100，50)处定义一个点。依次，我们在(450，400)处定义绿点，在(450，50)处定义蓝点。然后我们用glEnd结束画三 角形。但此时三角形还没画出来，这些命令还只是在缓冲区里，直到你调用glFlush函数，由glFlush触发这些命令的执行。OpenGL自动改变三 角形顶点间的颜色值，使之绚丽多彩。

还可通过glBegin再产生新的图形。glBegin(GLenum mode)参数有：
GL_POINTS,GL_LINES,GL_LINE_STRIP,GL_LINE_LOOP,
GL_TRIANGLES,GL_TRIANGLE_STRIP,GL_TRIANGLE_FAN,GL_QUADS, GL_QUAD_STRIP, GL_POLYGON

在glBegin和glEnd之间的有效函数有: glVertex,glColor,glIndex, glNormal,glTexCoord, glEvalCoord,glEvalPoint, glMaterial, glEdgeFlag

四．小结

1、如果要响应WM_SIZE消息，则一定要设置视口和矩阵模式。
2、尽量把你全部的画图工作在响应WM_PAINT消息时完成。
3、产生一个绘制环境要耗费大量的CPU时间，所以最好在程序中只产生一次，直到程序结束。
4、尽量把你的画图命令封装在文档类中，这样你就可以在不同的视类中使用相同的文档，节省你编程的工作量。
5、glBegin和glEnd一定要成对出现，这之间是对图元的绘制语句。

glPushMatrix()和glPopMatrix()也一定要成对出现。glPushMatrix()把当前的矩阵拷贝到栈中。当我们调用 glPopMatrix时，最后压入栈的矩阵恢复为当前矩阵。使用glPushMatrix()可以精确地把当前矩阵保存下来，并用 glPopMatrix把它恢复出来。这样我们就可以使用这个技术相对某个物体放置其他物体。例如下列语句只使用一个矩阵，就能产生两个矩形，并将它们成一定角度摆放。

```
glPushMatrix();
    glTranslated( m_transX, m_transY, 0);
    glRotated( m_angle1, 0, 0, 1);
    glPushMatrix();
    glTranslated( 90, 0, 0);
    glRotated( m_angle2, 0, 0, 1);
    glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
    glCallList(ArmPart);//ArmPart 且桓鬣卣竺?
    glPopMatrix();
    glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
    glCallList(ArmPart);
   glPopMatrix();
```

6、 解决屏幕的闪烁问题。我们知道，在窗口中拖动一个图形的时候，由于边画边显示，会出现闪烁的现象。在GDI中解决这个问题较为复杂，通过在内存中生成一个 内存DC，绘画时让画笔在内存DC中画，画完后一次用Bitblt将内存DC"贴"到显示器上，就可解决闪烁的问题。在OpenGL中，我们是通过双缓存 来解决这个问题的。一般来说，双缓存在图形工作软件中是很普遍的。双缓存是两个缓存，一个前台缓存、一个后台缓存。绘图先在后台缓存中画，画完后，交换到 前台缓存，这样就不会有闪烁现象了。通过以下步骤可以很容易地解决这个问题：

1) 要注意，GDI命令是没有设计双缓存的。我们首先把使用InvalidateRect(null)的地方改成InvalidateRect(NULL,FALSE)。这样做是使GDI的重画命令失效，由OpenGL的命令进行重画；
2) 将像素格式定义成支持双缓存的（注：PFD_DOUBLEBUFFER和PFD_SUPPORT_GDI只能取一个，两者相互冲突）。

```
pixelDesc.dwFlags =
PFD_DRAW_TO_WINDOW |
PFD_SUPPORT_OPENGL |
PFD_DOUBLEBUFFER |
PFD_STEREO_DONTCARE;
```

3) 我们得告诉OpenGL在后台缓存中画图，在视类的OnSize()的最后一行加入：glDrawBuffer (GL_BACK)；
4) 最后我们得把后台缓存的内容换到前台缓存中，在视类的OnPaint()的最后一行加入：SwapBuffers(dc.m_ ps.hdc)。

7、生成简单的三维图形。我们知道，三维和二维的坐标系统不同，三维的图形比二维的图形多一个z坐标。我们在生成简单的二维图形时，用的是 gluOrtho2D；我们在生成三维图形时，需要两个远近裁剪平面，以生成透视效果。实际上，二维图形只是视线的近裁剪平面z= -1，远裁剪平面z=1；这样z坐标始终当作0，两者没有本质的差别。

在上述基础之上，我们只做简单的变化，就可以生成三维物体。

1) 首先，在OnSize()中，把gluOrtho2D(0.0, 500.0*aspect,0.0, 500.0)换成gluPerspective(60, aspect, 1, 10.0)；这样就实现了三维透视坐标系的设置。该语句说明了视点在原点，透视角是60度，近裁剪面在z=1处，远裁剪面在z=10.0处。
2) 在RenderScene()中生成三维图形；实际上，它是由多边形组成的。下面就是一个三维多边形的例子：

```
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, RedSurface)
glBegin(GL_POLYGON);
glNormal3d( 1.0, 0.0, 0.0);
glVertex3d( 1.0, 1.0, 1.0);
glVertex3d( 1.0, -1.0, 1.0);
glVertex3d( 1.0, -1.0, -1.0);
glVertex3d( 1.0, 1.0, -1.0);
glEnd();
```

3) 我们使用glMaterialfv(GL_ FRONT_AND_BACK, GL_AMBIENT, RedSurface)这个函数来定义多边形的表面属性，为每一个平面的前后面设置环境颜色。当然，我们得定义光照模型，这只需在OnSize()的最后 加上glEnable(GL_LIGHTING)；RedSufFace是一个颜色分量数组，例如：RedSufFace[] ={1.0f,0.0f,0.0f}；要定义某个平面的环境颜色，只需把glMaterialfv加在平面的定义前面即可，如上例所示。

4) Z缓冲区的问题：要使三维物体显得更流畅，前后各面的空间关系正确，一定得使用Z缓冲技术；否则，前后各面的位置就会相互重叠，不能正确显示。Z缓冲区存 储物体每一个点的值，这个值表明此点离人眼的距离。Z缓冲需要占用大量的内存和CPU时间。启用Z缓冲只需在OnSize()的最后加上glEnable (GL_DEPTH_TEST)；要记住：在每次重绘之前，应使用glClear(GL_DEPTH_BUFFER_BIT)语句清空Z缓冲区。
5) 现在已经可以正确地生成三维物体了，但还需要美化，可以使物体显得更明亮一些。我们用glLightfv函数定义光源的属性值。下例就定义了一个光源：

```
glLightfv(GL_LIGHT0, GL_AMBIENT,LightAmbient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, LightDiffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, LightSpecular);
glLightfv(GL_LIGHT0, GL_POSITION, LightPosition);
glEnable(GL_LIGHT0);
```

GL_LIGHT0是光源的标识号，标识号由GL_LIGHTi组成(i从0到GL_MAX_LIGHTS)。GL_AMBIENT、GL_DIFFUSE、GL_SPECULAR、GL_POSITION分别定义光源的周围颜色强度、光源的散射强度、光源的镜面 反射强度和光源的位置。

作者：洞庭散人
出处：http://phinecos.cnblogs.com/