第一章 OpenGL 介绍

OpenGL 是近几年发展起来的一个性能卓越的三维图形标准，它是在 SGI 等多家世界闻名的计算机公司的倡导下，以 SGI 的 GL 三维图形库为基础制定的一个通用共享的开放式三维图形标准。目前，包括 Microsoft、SGI、IBM、DEC、SUN、HP 等大公司都采用了 OpenGL 做为三维图形标准，许多软件厂商也纷纷以 OpenGL 为基础开发出自己的产品，其中比较著名的产品包括动画制作软件 Soft Image 和 3D Studio MAX、仿真软件 Open Inventor、VR 软件 World Tool Kit、CAM 软件 ProEngineer、GIS 软 ARC/INFO 等等。

第一节 OpenGL 特点及功能

OpenGL 实际上是一个开放的三维图形软件包，它独立于窗口系统和操作系统，以它为基础开发的应用程序可以十分方便地在各种平台间移植；OpenGL 可以与 Visual C++紧密接口，可保证算法的正确性和可靠性；OpenGL 使用简便，效率高。它具有七大功能：

1. 建模

OpenGL 图形库除了提供基本的点、线、多边形的绘制函数外，还提供了复杂的三维物体（球、锥、多面体、茶壶等）以及复杂曲线和曲面（如 Bezier、Nurbs 等曲线或曲面）绘制函数。

2. 变换

OpenGL 图形库的变换包括基本变换和投影变换。基本变换有平移、旋转、变比镜像四种变换，投影变换有平行投影（又称正射投影）和透视投影两种变换。其变换方法与机器人运动学中的坐标变换方法完全一致，有利于减少算法的运行时间，提高三维图形的显示速度。

3. 颜色模式设置

OpenGL 颜色模式有两种，即 RGBA 模式和颜色索引(Color Index)。

4. 光照和材质设置

OpenGL 光有辐射光(Emitted Light)、环境光(Ambient Light)、漫反射光(Diffuse Light)和镜面光(Specular Light)。材质是用光反射率来表示。场景(Scene)中物体最终反映到人眼的颜色是光的红绿蓝分量与材质红绿蓝分量的反射率相乘后形成的颜色。

5. 纹理映射(Texture Mapping)

利用 OpenGL 纹理映射功能可以十分逼真地表达物体表面细节。

6. 位图显示和图象增强

图象功能除了基本的拷贝和像素读写外，还提供融合(Blending)、反走样(Antialiasing)和雾(fog)的特殊图象效果处理。以上三条可是被仿真物更具真实感，增强图形显示的效果。

7. 双缓存(Double Buffering)动画

双缓存即前台缓存和后台缓存，简而言之，后台缓存计算场景、生成画面，前台缓存显示后台缓存已画好的画面。此外，利用 OpenGL 还能实现深度暗示(Depth Cue)、运动模糊(Motion Blur)等特殊效果。从而实现了消隐算法。

第二节 OpenGL 的图形库

OpenGL 图形库一共有 100 多个函数。其中核心函数有 115 个，它们是最基本的函数，其前缀是 gl，OpenGL 实用库(OpenGL utility library ,GLU)的函数功能更高一些，如绘制复杂的曲线曲面、高级坐标变换、多边形分割等，共有 43 个，前缀为 glu；OpenGL 辅助库(OpenGL auxiliary library ,GLAUX)的函数是一些特殊的函数，包括简单的窗口管理、输入事件处理、某些复杂三维物体绘制等函数，共有 31 个，前缀为 aux。

此外，还有六个 WGL 函数非常重要，专门用于 OpenGL 和 Windows 窗口系统的联接，其前缀

为 wgl，主要用于创建和选择图形操作描述表(rendering contexts)以及在窗口内任一位置显示字符位图。

另外，还有五个 Win32 函数用来处理像素格式(pixel formats)和双缓存。由于它们是对 Win32 系统的扩展，因此不能应用在其它 OpenGL 平台上。

第二章 VC＋＋6.0 中使用 OpenGL

OpenGL for Windows 的设计与 OpenGL for UNIX 的程序设计有一点小区别，关键就在于如何将 OpenGL 与不同的操作系统下的窗口系统联系起来。如果调用 OpenGL 辅助库窗口管理函数，则不用考虑这些问题。

第一节 初始化设置

1. 图形操作描述

在 Windows 下，窗口程序必须首先处理设备描述表(Device Contexts, DC)，DC 包括许多如何在窗口上显示图形的信息，既指定画笔和刷子的颜色，设置绘图模式、调色板、映射模式以及其它图形属性。同样，OpenGL for Windows 的程序也必须使用 DC，这与其它 Windows 程序类似。但是，OpenGL for Windows 必须处理特殊的 DC 图形操作描述表，这是 DC 中专为 OpenGL 使用的一种。一个 OpenGL 应用图形操作描述表内有 OpenGL 与 Windows 窗口系统相关的各种信息。一个 OpenGL 应用首先必须创建一个图形操作描述表，然后再启动它，最后在所定义的窗口内按常规方式调用 OpenGL 函数绘制图形。

OpenGL 的图形操作描述表不同于其它 DC，其它 DC 调用每个 GDI 函数都需要一个句柄，而图形操作描述表方式下只需一个句柄就可以任意调用 OpenGL 函数。也就是说，只要当前启用了某个图形操作描述表，那么在未删除图形操作描述表之前可以调用任何 OpenGL 函数，进行各种操作。

2. 像素格式

在创建一个图形操作表之前，首先必须设置像素格式。像素格式含有设备绘图界面的属性，这些属性包括绘图界面是用 RGBA 模式还是颜色表模式，像素缓存是用单缓存还是双缓存，以及颜色位数、深度缓存和模板缓存所用的位数，还有其它一些属性信息。

3. 像素格式结构

OpenGL 显示设备都支持一种指定的像素格式。一般用一个名为 PIXELFORMATDESCRIPTOR 的结构来表示某个特殊的像素格式，这个结构包含 26 个属性信息。如下所示：

```
typedef struct tagPIXELFORMATDESCRIPTOR
{ // pfd
    WORD    nSize;
    WORD    nVersion;
    DWORD dwFlags;
    BYTE    iPixelType;
    BYTE    cColorBits;
    BYTE    cRedBits;
    BYTE    cRedShift;
    BYTE    cGreenBits;
    BYTE    cGreenShift;
    BYTE    cBlueBits;
    BYTE    cBlueShift;
    BYTE    cAlphaBits;
    BYTE    cAlphaShift;
    BYTE    cAccumBits;
    BYTE    cAccumRedBits;
    BYTE    cAccumGreenBits;
    BYTE    cAccumBlueBits;
    BYTE    cAccumAlphaBits;
    BYTE    cDepthBits;
    BYTE    cStencilBits;
    BYTE    cAuxBuffers;
```

```
    BYTE    iLayerType;
    BYTE    bReserved;
    DWORD dwLayerMask;
    DWORD dwVisibleMask;
    DWORD dwDamageMask;
} PIXELFORMATDESCRIPTOR;
```

4. 初始化 PIXELFORMATDESCRIPTOR 结构

PIXELFORMATDESCRIPTOR 结构中每个变量值的具体含义和设置可以参考有关资料，下面举出一个对 PIXELFORMATDESCRIPTOR 进行初始化的例子来简要说明相关变量的意义。定义 PIXELFORMATDESCRIPTOR 结构的 pfd 如下：

```
PIXELFORMATDESCRIPTOR pfd =
{
    sizeof(PIXELFORMATDESCRIPTOR),      //  size of this pfd
    1,
    PFD_DRAW_TO_WINDOW |                //  support window
    PFD_SUPPORT_OPENGL |                // support OpenGL
    PFD_DOUBLEBUFFER,                   //  double buffered
    PFD_TYPE_RGBA,                      // RGBA type
    24,                                 //  24-bit color depth
    0,  0,  0,  0,  0,  0,              //  color bits ignored
    0,                                  //  no alpha buffer
    0,                                  // shift bit ignored
    0,                                  //  no accumulation buf
    0, 0, 0, 0,                         // accum bits ignored
    32,                                 //  32-bit z-buffer
    0,                                  //  no stencil buffer
    0,                                  //  no auxiliary buffer
    PFD_MAIN_PLANE,                     // main layer
    0,                                  //  reserved
    0,  0,  0                           // layer masks ignored
};
```

该结构前两个变量的含义十分明显。第三个变量 dwFlags 的值是 PFD_DRAW_TO_WINDOW |PFD_SUPPORT_OPENGL|PFD_DOUBLEBUFFER，表明应用程序使用 OpenGL 函数来绘制窗口。第四个变量表明当前采用 RGBA 颜色模式。第五个变量采用 24 位真彩色，如果是 256 色系统则自动实现颜色抖动。因为没有使用 alpha 缓存和累计缓存，所以从变量 cAlphaBits 到 cAccumAlphaBits 都设置为 0。深度缓存设置为 32 位，这个缓存能解决三维场景的消隐问题。变量 cAuxBuffers 设为 0，Windows 下不支持辅助缓存。Windows 下变量 ilayerType 只能设置为 PFD_MAIN_PLANE，但在其它平台也许支持 PFD_MAIN_PLANE 或 PFD_MAIN_UNDERLAYPLANE。接下来 bReserved 变量只能设为 0，而最后三个变量 Windows 都不支持，故全设置为 0。

5. 设置像素结构

初始化 PIXELFORMATDESCRIPTOR 结构后，要设置像素格式。下面举例说明设置像素格式。

```
CClientDC    clientDC(this);
int          PixelFormat = ChoosePixelFormat(clientDC.m_hDC,&pfd);
BOOL         result=SetPixelFormat(clientDC.m_hDC,PixelFormat,&pfd);
```

第一行语句得到一个应用窗口客户区的设置描述表。

第一行调用 ChoosePixelFormat()选择一个像素格式，并将像素格式索引号返回给 pixelFormat 变量；函数中第一个参数是选择像素格式的设备描述表的句柄，第二个参数是 PIXELFORMATDESCRIPTOR 结构的地址。如果调用失败则返回 0；否则返回像素格式索引号。

第三行调用 SetPixelFormat()设置像素格式，三个参数分别是设备描述表的句柄、像素格式索引号和 PIXELFORMATDESCRIPTOR 结构的地址。如果调用成功则返回 TRUE，否则返回 FALSE。

6. 创建图形操作描述表

必须创建并启用图形操作描述表后，才能调用 OpenGL 函数在窗口内进行各种图形操作。一般来说，利用 MFC 中增补的管理图形操作描述表方法来编程比较方便。即在视类(CView)的消息 OnCreat()中创建图形操作描述表。

第二节 编程实例

该实例将建立一个视窗程序，以说明 OpenGL 图形的最小需求。这一任务将分 5 步来进行：设置窗口像素的格式；建立 RC；使 RC 设为当前；创建视口和矩阵模型；画一个立方体和一个茶壶。

1.打开 VisualC＋＋，建立一个单文档的项目。

2.在项目中加进所有必需的 OpenGL 文件和库。在菜单中选择 BuildSettings，点击 LINK 按钮，在 Object/Library 栏中键入 OpenGL32.lib; GLu32.lib; glaux.lib 并确定。打开文件 stdafx.h 插入如下行：

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

3.编辑 OnPreCreate 函数，指定窗口类型。OpenGL 仅能在具有 WS_CLIPCHILDREN 和 WS_CLIPSIBLINGS 类型的窗口显示图形。

```
BOOLCOPView :: PreCreateWindow ( CREATESTRUCT ＆cs )
{
        cs.style |= ( WS_CLIPCHILDREN | WS_CLIPSIBLINGS ) ;
        return CView :: PreCreateWindow ( cs ) ;
}
```

4.定义窗口的像素格式。首先建立一个受保护的成员函数 SetWindowPixelFormat。如下所示：

```
BOOL COPView :: SetWindowPixelFormat ( HDC hDC )
{
    PIXELFORMATDESCRIPTOR pixelDesc ;
    pixelDesc.nSize = sizeof ( PIXELFORMATDESCRIPTOR ) ;
    pixelDesc.nVersion = 1 ;
    pixelDesc.dwFlags = PFD_DRAW_TO_WINDOW |
                        PFD_DRAW_TO_BITMAP |
                        PFD_SUPPORT_OPENGL |
                        PFD_SUPPORT_GDI |
                        PFD_STEREO_DONTCARE ;
    pixelDesc.iPixelType = PFD_TYPE_RGBA ;
    pixelDesc.cColorBits = 32 ;
    pixelDesc.cRedBits = 8 ;
    pixelDesc.cRedShift = 16 ;
    pixelDesc.cGreenBits = 8 ;
    pixelDesc.cGreenShift = 8 ;
    pixelDesc.cBlueBits = 8 ;
    pixelDesc.cBlueShift = 0 ;
    pixelDesc.cAlphaBits = 0 ;
    pixelDesc.cAlphaShift = 0 ;
    pixelDesc.cAccumBits = 64 ;
    pixelDesc.cAccumRedBits = 16 ;
    pixelDesc.cAccumGreenBits = 16 ;
    pixelDesc.cAccumBlueBits = 16 ;
    pixelDesc.cAccumAlphaBits = 0 ;
    pixelDesc.cDepthBits = 32 ;
    pixelDesc.cStencilBits = 8 ;
    pixelDesc.cAuxBuffers = 0 ;
    pixelDesc.iLayerType = PFD_MAIN_PLANE ;
    pixelDesc.bReserved = 0 ;
    pixelDesc.dwLayerMask = 0 ;
    pixelDesc.dwVisibleMask = 0 ;
    pixelDesc.dwDamageMask = 0 ;

    m_GLPixelIndex = ChoosePixelFormat( hDC , ＆pixelDesc ) ;
    if ( m_GLPixelIndex == 0 )
```

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

5. 加入一个成员变量到视类中：

```
    int m_GLPixelIndex ;       //protected
```

6. 在 ClassWizard 中加入函数 OnCreate 来响应消息 WM_CREATE，函数如下：

```
    int COPView :: OnCreate ( LPCREATESTRUCT lpCreateStruct )
    {
        if ( CView :: OnCreate ( lpCreateStruct ) ==  －1 )
            return  －1 ;
        HWND hWnd = GetSafeHwnd ( ) ;
        HDC hDC = :: GetDC ( hWnd ) ;
        if ( SetWindowPixelFormat ( hDC ) == FALSE )
            return 0 ;
        if ( CreateViewGLContext ( hDC ) == FALSE )
            return 0 ;
        return 0 ;
    }
```

7. 建立 RC，并置为当前 RC。在视类中加入保护函数 CreateViewGLContext (HDC hDC)和变量 HGLRC m_hGLContext。

```
    BOOL COPView :: CreateViewGLContext (HDC hDC)
    {
        m_hGLContext = wglCreateContext ( hDC ) ;
        if ( m_hGLContext == NULL )
        {     return FALSE ;     }
        if ( wglMakeCurrent ( hDC , m_hGLContext ) == FALSE )
        {     return FALSE ;    }
        return    TRUE ;
    }
```

8. 加入函数 OnDestroy 来响应 WM_DESTROY:

```
    voidCOPView :: OnDestroy ( )
    {
        if ( wglGetCurrentContext ( ) != NULL )
        {       wglMakeCurrent ( NULL , NULL ) ;       }
        if ( m_hGLContext != NULL )
        {
            wglDeleteContext ( m_hGLContext ) ;
            m_hGLContext = NULL ;
        }
        CView :: OnDestroy( ) ;
    }
```

9. 编辑 COPView 类构造函数：

```
    COPView::COPView()
    {
        m_hGLContext = NULL ;
        m_GLPixelIndex = 0 ;
    }
```

10. 建立视点和矩阵模型。用 ClassWizard 在视类中加入函数 OnSize 响应 WM_SIZE。

```
    void COPView :: OnSize ( UINT nType , int cx , int cy )
    {
        CView :: OnSize ( nType , cx , cy ) ;
        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 ( ) ;
        gluPerspective ( 45 , aspect , 1 , 10.0 ) ;
        glMatrixMode ( GL_MODELVIEW ) ;
        glLoadIdentity ( ) ;
    }
```

11. 加入函数 OnPaint：

```
    void COPView :: OnPaint ( )
    {
        CPaintDC dc  ( this  ) ;
        COPDoc *pDoc = GetDocument ( ) ;
        pDoc  －> RenderScene ( ) ;
    }
```

12. 在文档类中加入公共函数 RenderScene()：

```
    void COPDoc :: RenderScene ( void )
    {
        glClear( GL_COLOR_BUFFER_BIT ) ;
        glFlush ( ) ;
    }
```

13. 在文档类中加一个枚举变量 GLDisplayListNames：

```
    enum GLDisplayListNames {ArmPart1 , ArmPart2 } ;
```

为将来建立显示列表用。

14. 编辑函数 OnNewDocument ( )：

```
    BOOL COPDoc :: OnNewDocument ( )
    {
        if( ! CDocument :: OnNewDocument ( ) )
            return FALSE ;
        glNewList ( ArmPart1 , GL_COMPILE ) ;
          GLfloat RedSurface [ ] = { 1.0f , 0.0f , 0.0f , 1.0f } ;
          GLfloat GreenSurface [ ] = { 0.0f , 1.0f , 0.0f , 1.0f } ;
          GLfloat BlueSurface [ ] = {0.0f , 0.0f , 1.0f , 1.0f } ;
          GLfloat LightAmbient [ ] = { 0.1f , 0.1f , 0.1f , 0.1f } ;
          GLfloat LightDiffuse [ ] = { 0.7f , 0.7f , 0.7f , 0.7f } ;
          GLfloat LightSpecular [ ] = { 0.0f , 0.0f , 0.0f , 0.1f } ;
          GLfloat LightPosition [ ] = { 5.0f , 5.0f , 5.0f , 0.0f } ;
          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 ) ;
          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  );       //画第一个面
          glEnd ( ) ;
          glBegin ( GL_POLYGON ) ;
            glNormal3d ( -1.0 , 0.0 , 0.0 ) ;
            此处同上画第二个面。
```

```
                立方体的中心为坐标原点。
            glEnd ( ) ;
            glMaterialfv ( GL_FRONT_AND_BACK , GL_AMBIENT , GreenSurface ) ;
            此处同上画第三、四个面，注意平面法向和坐标。
            glMaterialfv (GL_FRONT_AND_BACK , GL_AMBIENT , BlueSurface ) ;
            此处同上画第五、六个面。
        glEndList ( )   ;
        glNewList ( ArmPart2 , GL_COMPILE ) ;
            glMaterialfv ( GL_FRONT_AND_BACK , GL_AMBIENT , GreenSurface ) ;
            auxSolidTeapot  ( 10  ) ;          //用辅助库函数画茶壶。
            glEndList ( ) ;
        return TRUE ;
    }
```

15. 显示。编辑 RenderScene 函数：

```
    void COPDoc :: RenderScene ( void )
    {
        double m_angle1 = 60.0 ;
        double m_angle2 = 30.0 ;
        glClear ( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT ) ;
        glPushMatrix ( ) ;
        glTranslated ( 3.0 , 0.0 , -8.0 ) ;
        glRotated ( m_angle1 , 0 , 0 , 1 ) ;
        glRotated ( m_angle2 , 0 , 1 , 0 ) ;
        glCallList ( ArmPart1);
        glPopMatrix ( ) ;
        glPushMatrix ( ) ;
        glTranslated ( 0.0 , 0.0 , -8.0 ) ;
        glCallList ( ArmPart2 ) ;
        glPopMatrix ( ) ;
        glFlush ( ) ;
    }
```

VC 中 OpenGL 编程的步骤：

1．使用 AppWizard 创建应用程序框架，在文档和视结构中，视负责窗口中内容的显示，因此所有的 OpenGL 绘制工作应放在视类进行；

2．利用 ClassWizard 给视类添加如下成员函数：

重载 PreCreateWindow 函数；

响应 WM_CREATE 消息的 OnCreate 函数；

响应 WM__SIZE 消息的 OnSize 函数；

响应 WM__ERASEBKGND 消息的 OnEraseBKgnd 函数；

响应 WM__DESTROY 消息的 OnDestroy 函数；

重载 OnInitialUpdate 函数

3．在 PreCreateWindow 函数中加入如下语句：

cs.style=cs.style|WS_CLIPSIBLINGS|WS_CLIPCHILDREN；

窗口必须设置以上属性，否则像素格式就不能正确设置；

4．设置像素格式，创建绘制描述表。这一过程放在响应 WM_CREATE 消息的 OnCreate 函数中。WM_CREATE 消息窗口被创建时产生。

像素格式定义显示设备的三十特性，比如颜色位平面的数量和组织方式（RGB 模式还是索引模式），是否采用双缓存模式等等。OpenGl 和 GDI（Windows 的图形设置接口）的像素格式不同，而 Windows 应用中窗口的缺省设置是 GDI 像素格式，所以需要为执行 OpenGL 绘制任务的窗口专门指

定像素格式。对每个窗口，像素格式只能设置一次。

绘制描述表与设置描述表类似，但包含了更多有关 OpenGL 绘制信息。OpenGL 依赖绘制描述表操作显示硬件，因此调用 OpenGL 命令前，必须创建描述表并使其成为当前的绘制描述表。

设置像素格式、创建绘制描述是每个窗口进行 OpenGl 绘制前必不可少的步骤，所以放在 OnCreate 函数中最为合适。

5．响应 WM_SIZE 消息。当视窗的大小改变时，需发送 WM_SIZE 消息。在 WM_SIZE 的消息响应函数中可以获知改变后窗口的大小，因此在其中可以调用 glViewport 对场景的窗口变换进行更改以后适应窗口大小的改变，还可通过 glFrustum 等函数重新设置投影变换，使得当窗口的大小发生改变时，显示在窗口中的场景不会发生扭曲。

6．响应 WM_ERASEBKGND 消息。当需要重新设置窗口背景时，产生 WM_ERASEBKGND 消息。处理该消息的缺省操作是用当前背景色填充整个窗口。处理函数做适当改变：注释掉 OnEraseBKgnd 函数所有的语句，添加 returnTRUE，使该函数不执行操作，仅返回 TRUE 值。

7．重载 OnInitUpdate 函数。OnInitUpdate 函数为视对象专有，在视第一次显示之前，文档和视的结构刚刚被建起之后调用，此时，从视对象中可以获得与之相连的文档对象中的信息。根据文档和视的关系，OpenGL 的绘制信息一般通过文档对象存取，通过视对象，因此，在 OnInitUpdate 中适合做 OpenGL 的绘制前的初始化工作。初始化工作包括创建列表(显示列表索引作为视类对象的成员变量保存)。设置光照参数，装载纹理映射贴图，确定融合方式及参数，设置雾化参数以及其它绘制场景前需完成的工作。必要时可设置时钟，程序在以后的运行中将定时接收到 WM_TIMER 消息，只要在该消息的响应函数中有规律地改变场景设置，并使视图重新绘制，就可以在窗口中形成动画效果。

8．在 OnDraw 函数中绘制 OpenGL 场景。所有的绘制工作都应放在 OnDraw 函数中。

9．响应 WM_TIMER 消息。WM_TIMER 消息由程序中设定的时钟产生。若未设定时钟，则不用响应 WM_TIMER 消息。在 WM_TIMER 的消息处理函数中，更新场景，然后发送 WM_PAINT 消息或直接调用 OnDraw 函数，使窗口中显示改变后的场景。如此反复，形成动画。

10．响应 WM_DESTROY 消息。WM_DESTROY 消息在窗口销毁时产生。在 WM_DESTROY 的消息的处理函数中删除描述表。

必要时重载基本应用框架中应用程序类的 OnIdle 函数。OnIdle 函数在程序的消息队列为空时被调用，该函数执行的功能与 gluIdleFunc 函数设定的空闲回调函数的功能相似，即执行一些后台任务。

用 VisualC++6 实现 OpenGL 编程

一、OpenGL 简介

众所周知，OpenGL 原先是 SiliconGraphicsIncorporated（SGI 公司）在他们的图形工作站上开发高质量图像的接口。但最近几年它成为一个非常优秀的开放式三维图形接口。实际上它是图形软件和硬件的接口，它包括有 120 多个图形函数，"GL"是"GRAPHICLIBRARY"的缩写，意思是“图形库”。OpenGL 的出现使大多数的程序员能够在 PC 机上用 C 语言开发复杂的三维图形。微软在 VisualC++5 中已提供了三个 OpenGL 的函数库(glu32.lib,glau.lib,OpenGL32.lib)，可以使我们方便地编程，简单、快速地生成美观、漂亮的图形。例如，WindowsNT 中的屏幕保护程序中的花篮和迷宫等都给人们留下了深刻的印象。

二、生成 OpenGL 程序的基本步骤和条件

本文将给出一个例子，这个例子是一个用 OpenGL 显示图像的 Windows 程序，通过这个程序我们也可以知道用 OpenGL 编程的基本要求。我们知道，GDI 是通过设备句柄（DeviceContext 以下简称"DC"）来绘图，而 OpenGL 则需要绘制环境（RenderingContext，以下简称"RC"）。每一个 GDI 命令需要传给它一个 DC，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，所有在此线程中的 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。本例将首先产生一个 OpenGLRC 并使之成为当前 RC，

分为三个步骤：设置窗口像素格式；产生 RC；设置为当前 RC。

1、首先创建工程

用 AppWizard 产生一个 EXE 文件，选择工程目录，并在工程名字中输入"GLSample1"，保持其他的不变；第一步、选单文档(SDI)；第二步、不支持数据库；第三步、不支持 OLE；第四步、不选中浮动工具条、开始状态条、打印和预览支持、帮助支持的复选框（选中也可以，本文只是说明最小要求），选中三维控制(3DControls)；第五步、选中产生源文件注释并使用 MFC 为共享动态库；第六步、保持缺省选择。按 Finish 结束，工程创建完毕。

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/LibraryModules 的编辑框中输入"OpenGL32.libglu32.libglaux.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"，将下列语句插入到文件中（划下划线的语句为所加语句）：

```
#defineVC_EXTRALEAN//Excluderarely-usedstufffromWindowsheaders
#include<afxwin.h>//MFCcoreandstandardcomponents
#include<afxext.h>//MFCextensions
#include<gl\gl.h>
#include<gl\glu.h>
#ifndef_AFX_NO_AFXCMN_SUPPORT
#include<afxcmn.h>//MFCsupportforWindows95CommonControls
#endif//_AFX_NO_AFXCMN_SUPPORT
```

3、改写 OnPreCreate 函数并给视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：

```
BOOLCGLSample1View::PreCr-eateWindow(CREATESTRUCT&cs)
{
cs.style|=(WS_CLIPCHI-LDREN|WS_CLIPSIBLINGS);
returnCView::PreCreate-Window(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 CGLSample1View 的类中添加一个保护型的成员函数 BOOLSetWindowPixel-Format(HDChDC)（用鼠标右键添加），并编辑其中的代码，见程序 1。

```
BOOLCGLSample1View::SetWindowPixelFormat(HDChDC)
{
PIXELFORMATDESCRIPTORpixelDesc;
pixelDesc.nSize=sizeof(PIXELFORMATDESCRIPTOR);
pixelDesc.nVersion=1;
pixelDesc.dwFlags=PFD_DRAW_TO_WINDOW|
PFD_DRAW_TO_BITMAP|
PFD_SUPPORT_OpenGL|
PFD_SUPPORT_GDI|
PFD_STEREO_DONTCARE;
pixelDesc.iPixelType=PFD_TYPE_RGBA;
pixelDesc.cColorBits=32;
pixelDesc.cRedBits=8;
pixelDesc.cRedShift=16;
pixelDesc.cGreenBits=8;
pixelDesc.cGreenShift=8;
pixelDesc.cBlueBits=8;
pixelDesc.cBlueShift=0;
pixelDesc.cAlphaBits=0;
pixelDesc.cAlphaShift=0;
pixelDesc.cAccumBits=64;
pixelDesc.cAccumRedBits=16;
```

```
pixelDesc.cAccumGreenBits=16;
pixelDesc.cAccumBlueBits=16;
pixelDesc.cAccumAlphaBits=0;
pixelDesc.cDepthBits=32;
pixelDesc.cStencilBits=8;
pixelDesc.cAuxBuffers=0;
pixelDesc.iLayerType=PFD_MAIN_PLANE;
pixelDesc.bReserved=0;
pixelDesc.dwLayerMask=0;
pixelDesc.dwVisibleMask=0;
pixelDesc.dwDamageMask=0;

m_GLPixelIndex=ChoosePixelFormat(hDC,&pixelDesc);
if(m_GLPixelIndex==0)//Let'schooseadefaultindex.
{m_GLPixelIndex=1;
if(DescribePixelFormat(hDC,m_GLPixelIndex,
sizeof(PIXELFORMATDESCRIPTOR),&pixelDesc)==0)
{returnFALSE;
}
}
if(SetPixelFormat(hDC,m_GLPixelIndex,&pixelDesc)==FALSE)
{returnFALSE;
}
returnTRUE;
}
```

接着用鼠标右键在 CGLSample1View 中添加保护型的成员变量：

```
intm_GLPixelIndex;
```

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

添加 OnCreate 函数后如程序 1 所示。

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 WindowsGDI 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 CGLSample1View 中加入一个保护型的成员函数 BOOLCreateViewGLContext(HDChDC)，使之如下所示：

```
BOOLCGLSample1View::CreateViewGLContext(HDChDC)
{m_hGLContext=wglCreateContext(hDC);//用当前 DC 产生绘制环境(RC)
if(m_hGLContext==NULL)
{returnFALSE;
}
if(wglMakeCurrent(hDC,m_hGLContext)==FALSE)
{returnFALSE;
}
returnTRUE;
}
```

并加入一个保护型的成员变量 HGLRCm_hGLContext；HGLRC 是一个指向 renderingcontext 的句柄。

在 OnCreate 函数中调用此函数：

```
intCGLSample1View::OnCreate(LPCREATESTRUCTlpCreateStruct)
{
if(CView::OnCreate(lpCreateStruct)==-1)
return-1;
HWNDhWnd=GetSafeHwnd();
HDChDC=::GetDC(hWnd);
if(SetWindowPixelFormat(hDC)==FALSE)
return0;
if(CreateViewGLContext(hDC)==FALSE)
return0;
return0;
}
```

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

```
voidCGLSample1View::OnDestroy()
{
if(wglGetCurrentContext()!=NULL)
{//maketherenderingcontextnotcurrent
wglMakeCurrent(NULL,NULL);
}
if(m_hGLContext!=NULL)
{wglDeleteContext(m_hGLContext);
m_hGLContext=NULL;
}
//NowtheassociatedDCcanbereleased.
CView::OnDestroy();
```

}

最后，编辑 CGLSample1View 的构造函数，使之如下所示：

```
CGLTutor1View::CGLTutor1View()
{m_hGLContext=NULL;
m_GLPixelIndex=0;
}
```

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。这与大多数的 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(DeviceContext)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、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设置为基础的）。

用 Classwizard 为 CGLSample2view 添加 WMSIZE 的消息处理函数 OnSize，使之如程序 2 所示。

（图注 getpwd2）图 2

用 Classwizard 为 CGLSample2view 添加 WM_PAINT 的消息处理函数 OnPaint，使之如程序 3 所示。

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（如图 2 所示）。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(projectmatrix)；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 OpenGLUtility。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(GLenummode)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、OpenGL 编程小结

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 且桓鼍卣竺□
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。

本文例子较简单，VisualC++6.0 中还有很多例子。参照本文的设置，你一定能体会到 OpenGL 强大的图形、图像绘制功能。

```
voidCGLSample2View::OnSize(UINTnType,intcx,intcy)
{
CView::OnSize(nType,cx,cy);
GLsizeiwidth,height;
GLdoubleaspect;
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();
}

voidCGLSample2View::OnPaint()
{CPaintDCDC(this);//devicecontextforpainting(addedbyClassWizard)
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();
}
```