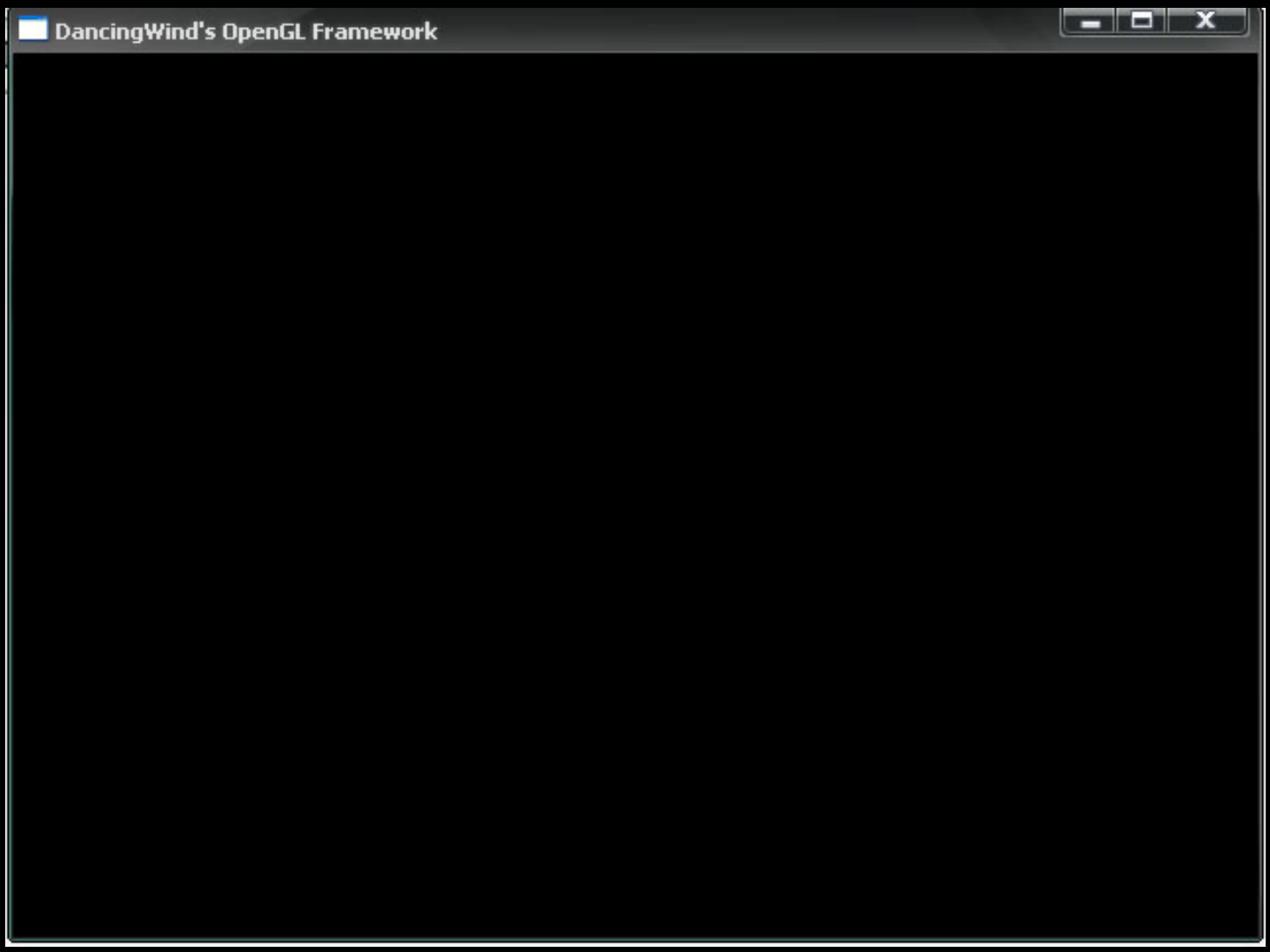
5、创建一个空白的OpenGL程序框架



NeHe SDK是把Nehe的教程中所介绍的所有功能，以面向对象的形式，提供给编程人员快速开发的一套编程接口。在下面的教程中，我将按NeHe SDK源码的功能分类，一步一步把这套api介绍给大家。如果你觉得有更好的学习方法，或者有其他有益的建议，请联系我。zhouwei02@mails.tsinghua.edu.cn, zhouwei506@mails.gucas.ac.cn

我在第四课的基础上，把公共的功能提取出来，并删除了特定的绘制函数，就完成了这个空白的程序框架，它可以使用基本的OpenGL绘制命令，并提供视口，文本，纹理类的接口，实用他们可以快速创建你想要的效果。

这个框架分为两个部分，启用main.cpp文件完成创建窗口，提供必需的全局变量和执行程序循环等固定的功能。在以后的演示程序中一般不在改变。draw.cpp文件完成具体的绘制操作，它随你的应用而变化。

main.cpp的所有功能在前面的课程中都详细讲解过，这里只是把整个程序结构在说明一下，让你有一个清

晰的认识。下面是它的七大结构

1. 头文件和全局变量
2. Windows主函数
3. 根据用户设置配置OpenGL的窗口
4. 创建OpenGL运行的窗口，并返回窗口的句柄
5. 设置绘制字体的参数
6. 程序循环
7. 退出程序

我们一一介绍如下：

1、头文件和全局变量

```
#include "opengl.h"                         // 包含创建OpenGL程序的框架类
#include "splash.h"                         // 创建配置对话框
#include "view.h"                           // 包含视口类的声明
#include "text.h"                           // 包含2D文字类的声明
#include "texture.h"                        // 包含纹理类的声明

#pragma comment( lib, "NeheSDK.lib" )       // 包含NeheSDK.lib库

using namespace NeHe;                       // 使用NeHe名字空间

View    view;                               // 创建视口类
Text2D  text2D;                             // 2D文字类
Texture tex;                                // 使用全局变量Texture类的实例
int     texID;                              // 使用全局变量texID，保存加载的纹理ID
```

2、Windows主函数

```
int WINAPI WinMain(HINSTANCE hInstance,     // 程序实例句柄
            HINSTANCE hPrevInstance, // 前一个程序实例句柄
            LPSTR     lpCmdLine,    // 命令行参数
            int       nCmdShow)     // Window 显示状态
{
    OpenGL  WinOpenGL;          // OpenGL类
```

3、根据用户设置配置OpenGL的窗口

```
/*****************************************************根据用户设置配置OpenGL的窗口
*****************************************************/
    // 显示配置对话框
    SplashResolution    res;                    // 记录分辨率
    SplashDepth             depth;      // 记录颜色深度
    bool                    fs;         // 是否全屏
    if(!DoSplash("setup.cfg",&res,&depth,&fs))
        return 1;

    int width,height;               // 窗口的大小
    int bpp;                        // 颜色位深

    // 设置分辨率
    switch(res)
    {
        case sr640x480:     width=640;    height=480; break;
        case sr800x600:     width=800;    height=600; break;
        case sr1024x768:            width=1024;   height=768; break;
        default:
            width=800; height=600;
    };

    // 设置颜色位深
    switch(depth)
    {
        case sd8bit:   bpp=8;  break;
        case sd16bit:  bpp=16; break;
        case sd32bit:  bpp=32; break;
        default:
            bpp=32;
    };

    // 设置是否全屏
    WinOpenGL.SetFullScreen((fs==true) ? true : false);
/*****************************************************根据用户设置配置OpenGL的窗口：结束
*****************************************************/
```

4、创建OpenGL运行的窗口，并返回窗口的句柄

```
/*****************************************************创建OpenGL运行的窗口，并返回窗口的句柄
*****************************************************/
    // 创建我们的OpenGL窗口
    if (!WinOpenGL.CreateGLWindow("DancingWind's OpenGL Framework", width, height, bpp, WinOpenGL.GetFullScreen
()))
    {
        return 0; // 失败，则退出
```

```
    }

    // 返回窗口类
    Window *win=WinOpenGL.GetWindow();
/*************************************************创建OpenGL运行的窗口，并返回窗口的句柄：结束
*****************************************/
```

5、设置绘制字体的参数

```
/*****************************************************设置绘制字体的参数
*********************************************************************/
    TextType ttype;                         // 设置字体结构
    ttype.name="Courier New";                       // 字体名称为"Courier New"
    ttype.size=24;                          // 字体大小为24
    ttype.bold=false;                               // 不使用粗体
    ttype.italic=false;                     // 不使用斜体
    ttype.underline=false;                      // 不使用下划线
    text2D.Setup(&WinOpenGL,ttype);             // 设置字体
/*****************************************************设置绘制字体的参数：结束
*********************************************************/
```

6、程序循环

```
/*************************************************************程序循环
*********************************************************************/
    bool finish=false;

    // 执行程序循环
    while(!finish)
    {
        // 绘制场景
        finish=!WinOpenGL.DrawGLScene();

        // 按ESC退出
        if(!finish)
            finish=win->GetKey(VK_ESCAPE);
    }
/*************************************************************程序循环：结束
*********************************************************/
```

7、退出程序

```
/***********************************************************退出程序
************************************************************/
  // 关闭
  WinOpenGL.KillGLWindow();

  return 0;  // 退出程序
/***********************************************************退出程序：结束
************************************************************/
}
```

到这里框架结构就完成了，下面我们进入到draw.cpp文件中。这个文件主要完成你特定功能的绘制操作，为了以后的演示方便，我们定义了一些默认函数，当你熟悉了以后，完全可以使用自己的函数替代这些功能简单的“玩具函数:)”。它主要提供了绘制函数的接口，完成以下四个功能：

1. 头文件和全局变量
2. 绘制三棱锥
3. 初始化场景
4. 设置默认的视口棱台体
5. 绘制场景

我们一一介绍如下：

1、头文件和全局变量

```
#include "opengl.h"                     // 包含创建OpenGL程序的框架类
#include "view.h"                       // 包含视口类的声明
#include "text.h"                       // 包含2D文字类的声明
#include "texture.h"                    // 包含纹理类的声明

#pragma comment( lib, "NeheSDK.lib" )   // 包含NeheSDK.lib库

using namespace NeHe;                   // 使用NeHe名字空间

extern View     view;                   // 使用全局变量View类的实例
extern Text2D   text2D;                 // 使用全局变量Text2D类实例
extern Texture  tex;                    // 使用全局变量Texture类的实例
extern int      texID;                  // 使用全局变量texID，保存加载的纹理ID

static bool initialize = true;          // 记录是否初始化
```

在上面的声明中，extern说明使用外部的全局变量，这里指的是实用main.cpp文件中声明的全局变量。

initialize变量用来记录是否在绘制过程中调用初始化函数，如果为true，则需要调用初始化函数一次。

2、绘制三棱锥函数

为了说明方便，我们定义两个绘制三棱锥的函数，DrawTri是实用固定颜色绘制三棱锥，DrawTexTri是使用纹理坐标绘制三棱锥。三棱锥在模型空间中的范围是：

X方向：-1到1
Y方向：-1到1
Z方向：0到1

记得让这个范围位于你的视口棱台体内，否则你将什么也看不到。

下面我们来看看具体的代码：

```
/*****************************************************绘制三棱锥
*****************************************************************************/
// 绘制三棱锥
void DrawTri(void)
{
    glPushAttrib(GL_CURRENT_BIT);            // 保存当前的绘制属性
    glBegin(GL_TRIANGLES);
        // 前面
        glColor3f(1.0f,0.0f,0.0f);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glColor3f(0.0f,1.0f,0.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);
        glColor3f(0.0f,0.0f,1.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);

        // 右面
        glColor3f(1.0f,0.0f,0.0f);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glColor3f(0.0f,0.0f,1.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);
        glColor3f(0.0f,1.0f,0.0f);
        glVertex3f( 1.0f,-1.0f, -1.0f);

        // 后面
        glColor3f(1.0f,0.0f,0.0f);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glColor3f(0.0f,1.0f,0.0f);
        glVertex3f( 1.0f,-1.0f, -1.0f);
        glColor3f(0.0f,0.0f,1.0f);
        glVertex3f(-1.0f,-1.0f, -1.0f);
```

```
        // 左面
        glColor3f(1.0f,0.0f,0.0f);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glColor3f(0.0f,0.0f,1.0f);
        glVertex3f(-1.0f,-1.0f,-1.0f);
        glColor3f(0.0f,1.0f,0.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);
    glEnd();
    glPopAttrib();                      // 弹出保存的绘制属性
}

// 绘制带纹理坐标的三棱锥
void DrawTexTri(void)
{
    glPushAttrib(GL_CURRENT_BIT);           // 保存当前的绘制属性
    glBegin(GL_TRIANGLES);
        // 前面
        glTexCoord2f(0.5f,0.5f);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glTexCoord2f(0.0f,0.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);
        glTexCoord2f(1.0f,0.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);

        // 右面
        glTexCoord2f(0.5f,0.5f);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glTexCoord2f(1.0f,0.0f);
        glVertex3f( 1.0f,-1.0f, 1.0f);
        glTexCoord2f(1.0f,1.0f);
        glVertex3f( 1.0f,-1.0f, -1.0f);

        // 后面
        glTexCoord2f(0.5f,0.5f);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glTexCoord2f(1.0f,1.0f);
        glVertex3f( 1.0f,-1.0f, -1.0f);
        glTexCoord2f(0.0f,1.0f);
        glVertex3f(-1.0f,-1.0f, -1.0f);

        // 左面
        glTexCoord2f(0.5f,0.5f);
        glVertex3f( 0.0f, 1.0f, 0.0f);
        glTexCoord2f(1.0f,1.0f);
        glVertex3f(-1.0f,-1.0f,-1.0f);
        glTexCoord2f(0.0f,0.0f);
        glVertex3f(-1.0f,-1.0f, 1.0f);
    glEnd();
    glPopAttrib();                      // 弹出保存的绘制属性
}
```

```
/*****************************************************绘制三棱锥:结束
*************************************************************************/
```

3、初始化场景

我们创建一个空的初始化函数，在你的应用中可以把那些在绘制开始时需要做的工作添加到这里面来。

```
/*****************************************************初始化场景
*************************************************************************/
void IniScene(OpenGL* gl, ControlData* cont)
{

}
/*****************************************************初始化场景:结束
*************************************************************************/
```

4、设置默认的视口棱台体

一般在绘制场景的开始，都要把模型变换矩阵和投影变换矩阵设置为单位矩阵。为了简化这个工程我们创建了这个函数。它创建的视口棱台体的范围如下：

- 默认的视口棱台体的范围是，视角45度，近切面距离视点0.1，远切面距离视点100。
- 默认在z=0的平面上的可见范围是，Y轴方向(-2,+2),X轴方向(-2,+2)*宽高比
- 视点位于(0,0,5),朝向-Z轴，上方向量为Y轴
- 宽高比由窗口大小决定，如果窗口大小为800x600，则宽高比为4/3，X轴的可见范围是(-2.67,+2.67)

```
/*****************************************************设置默认视口棱台体
*************************************************************************/
void SetDefaultView()
{
    // 默认的视口棱台体的范围是，视角45度，近切面距离视点0.1，远切面距离视点100。
    // 默认在z=0的平面上的可见范围是，Y轴方向(-2,+2),X轴方向(-2,+2)*宽高比
    // 宽高比由窗口大小决定，如果窗口大小为800x600，则宽高比为4/3，X轴的可见范围是(-2.67,+2.67)
    view.Reset();                                   // 重置视口
    view.LookAt(Vector(0,0,5),Vector(0,0,0),Vector(0,1,0));      // 设定视口在(0,0,5),朝向-Z轴，上方向量为Y轴
}
/*****************************************************设置默认视口棱台体：结束
*************************************************************************/
```

5、绘制场景

在创建第一个场景时，需要调用初始化场景的操作，每绘制一次场景都要把视口重新设定一下。故把这个功能加入到了绘制场景函数的开始部分，代码如下：

```
/*****************************************************绘制场景
*************************************************************************/
void DrawScene(OpenGL *gl,ControlData *cont)
{
    // 初始化
    if(initialize)
    {
        IniScene(gl,cont);
        initialize = false;
    }
    SetDefaultView();                                   // 设置默认的视口
}
/*****************************************************绘制场景:结束
*************************************************************************/
```

好了上面就是整个程序框架了，下面我们简单的回顾一下如何快速实用各个类提供的功能，主要包括以下三个功能：

1. 设置视口
2. 绘制文本
3. 设置纹理

1、设置视口

在程序中重新设置视口只需要下面两个步骤，重置视口，设置视点的位置和方向，不要忘了我们的视口棱台体的范围是：

- 默认的视口棱台体的范围是，视角45度，近切面距离视点0.1，远切面距离视点100。
- 默认在z=0的平面上的可见范围是，Y轴方向(-2,+2),X轴方向(-2,+2)*宽高比

下面是一个示例代码

```
view.Reset();                                   // 重置视口
view.LookAt(Vector(0,0,5),Vector(0,0,0),Vector(0,1,0));      // 设定视口在(0,0,5),朝向-Z轴，上方向量为Y轴
```

2、绘制文本

下面的代码告诉你如何在窗口的顶层绘制文本，默认文本在三维空间中位于z=0的平面，当然你可以使用Translate函数来改变它在空间中的位置。2D文本一直面对视点，Rotate函数对它不起作用。

```
view.Save();                                    // 保存当前的视口矩阵
view.Reset();                                   // 重置视口矩阵
view.Translate(0.0f,0.0f,-5.0f);                        // 把绘制的模型坐标向-Z轴移动5个单位
view.Pos2D(-2.5f,1.0f);                         // 设置在z=0平面，绘制点的位置
glPushAttrib(GL_DEPTH_BUFFER_BIT);                              // 保存深度缓存的属性
glDisable(GL_DEPTH_TEST);                               // 禁用深度测试，可以让我们的文本始终显示在最上层
text2D<<"Draw a pyramid with texture";                  // 绘制文本
glPopAttrib();                                  // 弹出保存的深度缓存属性
view.Restore();                                 // 弹出保存的视口矩阵
```

3、设置纹理

在OpenGL中使用纹理需要从文件中加载，并启用它，如果你不考虑效率的话，可以如下设置：

```
//载入base.bmp纹理
texID = tex.Load("base.bmp");
if(texID==0)
{
    MessageBox(NULL,"不能加载base.bmp图像","Error",MB_OK| MB_ICONEXCLAMATION);
    cont->quit=true;
    return;
}
//启用纹理并绘制带纹理坐标的模型
// 启用纹理
tex.Set(texID);
cont->state->SetTexturing(true);
// 绘制纹理多边形
DrawTexTri();
```

好了，这就使这一课的全部了，怎么样，现在对整个程序比较清楚了吧。最后在说一句，最好在初始化时载入纹理，这样你不用在绘制时反复载入文件，这样可以提高效率。

后面的演示实例都是建立在这一课代码的基础上了，并且都在draw.cpp中添加新的代码，所以这一课很关键，希望你能读懂它。