Qt OpenGL 教程

最近一段时间除了学习 Qt，翻译 Qt 文档之外，由于工作和兴趣的原因，开始着手看 Qt OpenGL 编程。在网上搜索了有关 OpenGL 的教程，发现 NeHe 的 OpenGL 教程的还很不错，作者是 NeHe。上面有很多种语言的实现，但是没有 Qt 和 Gtk 的，所以我就想着手写这个 Qt OpenGL 教程，每课的内容和 NeHe 是一样的。另外，介绍 NeHe 的一个中文翻译站点 CSDN-CKer 翻译的 NeHe 的 OpenGL 教程，翻译人是 CKer，在我学习这个教程的过程中，给了我很大的帮助。

下面就是 Qt OpenGL 教程的内容：

Qt OpenGL 的准备工作

第一课：创建一个 OpenGL 窗口

第二课：你的第一个多边形

第三课：上色

第四课：旋转

第五课：向三维进军

第六课：纹理映射

第七课：纹理滤波、光源和键盘控制

第八课：融合

第九课：在三维空间中移动位图

第十课：载入一个三维世界并在其中移动

第十一课：旗的效果（波动纹理）

第十二课：显示列表

第十三课：位图字体

第十四课：轮廓字体

第十五课：使用纹理映射的轮廓字体

第十六课：看起来很棒的雾

因为本教程是从 NeHe 的 OpenGL 教程迁移过来的，代码变为 Qt 实现的。所以有的课程一时还没有实现成功，所以可能有些教程是跳跃的。

因本人时间有限，所以难免有错误出现，如果您发现了这些错误，或者有什么建议，请来信指教，谢谢。

Qt OpenGL 的准备工作

因为 Qt 存在很多版本，另外它支持的平台也很多，到目前为止我只实验了几个组合，所以就先把这些列出来吧，欢迎大家补充。

Unix/X11

Linux

Qt：自由版或者企业版都支持 OpenGL 模块，而专业版则不能。我现在使用的是 3.1.0 自由版和企业版。

gcc：编译器。我现在使用的是 3.2。

X：Linux 下的图形环境。我现在使用的是 4.2.0。

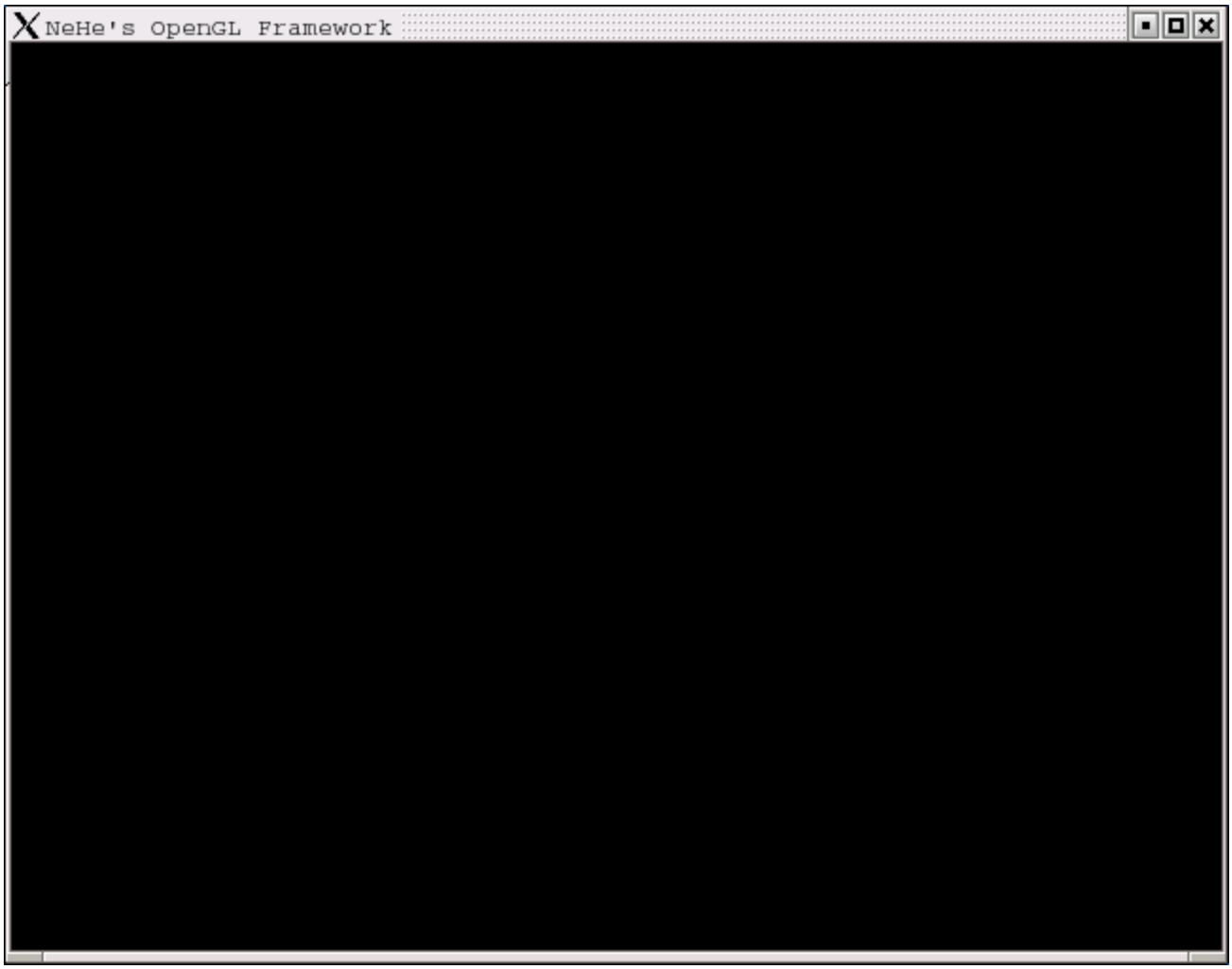
Mesa：自由的 OpenGL。我现在使用的是 5.0。

Windows

Qt：企业版支持 OpenGL 模块，而专业版则不能。我现在使用的是 3.1.0 企业版。

Microsoft Visual Studio：编译器。我现在使用的是 6.0。

创建一个 OpenGL 窗口



我假设您对 Qt 编程已经有了一定的了解，如果您还没有熟悉 Qt 编程，建议您先学习一下 Qt 编程的基础知识。

Qt 中已经包含了 OpenGL 模块，具体情况您可以参考 Qt OpenGL 模块的相关内容。

NeHeWidget 类

这就是我们继承 QGLWidget 类得到的 OpenGL 窗口部件类。

（由 **nehewidget.h** 展开。）

```
#include <qgl.h>

class NeHeWidget : public QGLWidget
{
    Q_OBJECT
```

因为 QGLWidget 类被包含在 qgl.h 头文件中，所以我们的类就需要包含这个头文件。Q_OBJECT 是 Qt 中的一个专用的宏，具体说明请参见 Qt 的文档。

```
public:

  NeHeWidget( QWidget* parent = 0, const char* name = 0, bool fs = false );
  ~NeHeWidget();

protected:

  void initializeGL();
  void paintGL();
  void resizeGL( int width, int height );
```

因为 QGLWidget 类已经内置了对 OpenGL 的处理，就是通过对 initializeGL()、paintGL()和 resizeGL()这个三个函数实现的，具体情况可以参考 QGLWidget 类的文档。

因为我们的这个 Qt OpenGL 教程取材于 NeHe OpenGL 教程，所以这里就用这个 NeHeWidget 类来继承 QGLWidget 类来使用相关 OpenGL 的功能。

initializeGL()是用来初始化这个 OpenGL 窗口部件的，可以在里面设定一些有关选项。paintGL()就是用来绘制 OpenGL 的窗口了，只要有更新发生，这个函数就会被调用。resizeGL()就是用来处理窗口大小变化这一事件的，width 和 height 就是新的大小状态下的宽和高了，另外 resizeGL()在处理完后会自动刷新屏幕。

```
  void keyPressEvent( QKeyEvent *e );
```

这是 Qt 里面的鼠标按下事件处理函数。

```
protected:
```

```
  bool fullscreen;
```

用来保存窗口是否处于全屏状态的变量。

```
};
```

（由 **nehewidget.cpp** 展开。）

```
#include "nehewidget.h"

NeHeWidget::NeHeWidget( QWidget* parent, const char* name, bool fs )
    : QGLWidget( parent, name )
{
  fullscreen = fs;
```

保存窗口是否为全屏的状态。

```
  setGeometry( 0, 0, 640, 480 );
```

设置窗口的位置，即左上角为(0,0)点，大小为 640*480。

```
  setCaption( "NeHe's OpenGL Framework" );
```

设置窗口的标题为“NeHe's OpenGL Framework”。

```
  if ( fullscreen )
    showFullScreen();
```

如果 fullscreen 为真，那么就全屏显示这个窗口。

```
}
```

这个是构造函数，parent 就是父窗口部件的指针，name 就是这个窗口部件的名称，fs 就是窗口是否最大化。

```
NeHeWidget::~NeHeWidget()
{
}
```

这个是析构函数。

```
void NeHeWidget::initializeGL()
{
  glShadeModel( GL_SMOOTH );
```

这一行启用 smooth shading(阴影平滑)。阴影平滑通过多边形精细的混合色彩，并对外部光进行平滑。我将在另一个教程中更详细的解释阴影平滑。

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

这一行设置清除屏幕时所用的颜色。如果您对色彩的工作原理不清楚的话，我快速解释一下。色彩值的范围从 0.0 到 1.0。0.0 代表最黑的情况，1.0 就是最亮的情况。glClearColor 后的第一个参数是红色,第二个是绿色，第三个是蓝色。最大值也是 1.0，代表特定颜色分量的最亮情况。最后一个参数是 Alpha 值。当它用来清除屏幕的时候，我们不用关心第四个数字。现在让它为 0.0。我会用另一个教程来解释这个参数。

通过混合三种原色（红、绿、蓝），您可以得到不同的色彩。希望您在学校里学过这些。因此，当您使用 glClearColor(0.0, 0.0, 1.0, 0.0)，您将用亮蓝色来清除屏幕。如果您用 glClearColor(0.5, 0.0, 0.0, 0.0)的话，您将使用中红色来清除屏幕。不是最亮(1.0)，也不是最暗 (0.0)。要得到白色背景，您应该将所有的颜色设成最亮(1.0)。要黑色背景的话，您该将所有的颜色设为最暗(0.0)。

```
  glClearDepth( 1.0 );
```

设置深度缓存。

```
  glEnable( GL_DEPTH_TEST );
```

启用深度测试。

```
  glDepthFunc( GL_LEQUAL );
```

所作深度测试的类型。

上面这三行必须做的是关于 depth buffer（深度缓存）的。将深度缓存设想为屏幕后面的层。深度缓存不断的对物体进入屏幕内部有多深进行跟踪。我们本节的程序其实没有真正使用深度缓存，但几乎所有在屏幕上显示 3D 场景 OpenGL 程序都使用深度缓存。它的排序决定那个物体先画。这样您就不会将一个圆形后面的正方形画到圆形上来。深度缓存是 OpenGL 十分重要的部分。

```
  glHint( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
```

真正精细的透视修正。这一行告诉 OpenGL 我们希望进行最好的透视修正。这会十分轻微的影响性能。但使得透视图看起来好一点。

```
}
```

这个函数中，我们对 OpenGL 进行所有的设置。我们设置清除屏幕所用的颜色，打开深度缓存，启用 smooth shading（阴影平滑），等等。这个例程直到 OpenGL 窗口创建之后才会被调用。

```
void NeHeWidget::paintGL()
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

清楚屏幕和深度缓存。

```
  glLoadIdentity();
```

重置当前的模型观察矩阵。

```
}
```

这个函数中包括了所有的绘图代码。任何您所想在屏幕上显示的东东都将在此段代码中出现。以后的每个教程中我都会在例程的此处增加新的代码。如果您对 OpenGL 已经有所了解的话，您可以在 glLoadIdentity()调用之后，函数返回之前，试着添加一些 OpenGL 代码来创建基本的形。如果您是 OpenGL 新手，等着我的下个教程。目前我们所作的全部就是将屏幕清除成我们前面所决定的颜色，清除深度缓存并且重置场景。我们仍没有绘制任何东东。

```
void NeHeWidget::resizeGL( int width, int height )
{
  if ( height == 0 )
  {
    height = 1;
  }
```

防止 height 为 0。

```
  glViewport( 0, 0, (GLint)width, (GLint)height );
```

重置当前的视口（Viewport）。

```
  glMatrixMode( GL_PROJECTION );
```

选择投影矩阵。

```
  glLoadIdentity();
```

重置投影矩阵。

```
  gluPerspective( 45.0, (GLfloat)width/(GLfloat)height, 0.1, 100.0 );
```

建立透视投影矩阵。

```
  glMatrixMode( GL_MODELVIEW );
```

选择模型观察矩阵。

```
  glLoadIdentity();
```

重置模型观察矩阵。

```
}
```

上面几行为透视图设置屏幕。意味着越远的东西看起来越小。这么做创建了一个现实外观的场景。此处透视按照基于窗口宽度和高度的 45 度视角来计算。0.1，100.0 是我们在场景中所能绘制深度的起点和终点。

glMatrixMode(GL_PROJECTION)指明接下来的两行代码将影响 projection matrix（投影矩阵）。投影矩阵负责为我们的场景增加透视。 glLoadIdentity()近似于重置。它将所选的矩阵状态恢复成其原始状态。调用 glLoadIdentity()之后我们为场景设置透视图。

glMatrixMode(GL_MODELVIEW)指明任何新的变换将会影响 modelview matrix（模型观察矩阵）。模型观察矩阵中存放了我们的物体讯息。最后我们重置模型观察矩阵。如果您还不能理解这些术语的含义，请别着急。在以后的教程里，我会向大家解释。只要知道如果您想获得一个精彩的透视场景的话，必须这么做。

这个函数的作用是重新设置 OpenGL 场景的大小，而不管窗口的大小是否已经改变（假定您没有使用全屏模式）。甚至您无法改变窗口的大小时（例如您在全屏模式下），它至少仍将运行一次——在程序开始时设置我们的透视图。OpenGL 场景的尺寸将被设置成它显示时所在窗口的大小。

```
void NeHeWidget::keyPressEvent( QKeyEvent *e )
{
  switch ( e->key() )
  {
  case Qt::Key_F2:
    fullscreen = !fullscreen;
    if ( fullscreen )
    {
      showFullScreen();
    }
    else
    {
      showNormal();
      setGeometry( 0, 0, 640, 480 );
    }
    updateGL();
    break;
```

如果按下了 F2 键，那么屏幕是否全屏的状态就切换一次。然后再根据需要，显示所要的全屏窗口或者普通窗口。

```
  case Qt::Key_Escape:
    close();
  }
```

如果按下了 Escape 键，程序退出。

```
}
```

main.cpp

（由 main.cpp 展开。）

```
#include <qapplication.h>
#include <qmessagebox.h>
```

Qt 的应用程序都是一个 QApplication 类，所以 qapplication.h 必须要包含。因为我们在进入 OpenGL 窗口之前让用户选择是否使用全屏窗口，所以使用了 QMessageBox 类，所以 qmessagebox.h 也要包含。

```
#include "nehewidget.h"
int main( int argc, char **argv )
{
  bool fs = false;
```

我们把这个布尔型变量的初始值设置为 false。

```
  QApplication a(argc,argv);
```

每一个 Qt 应用程序都使用 QApplication 类。

```
  switch( QMessageBox::information( 0,
      "Start FullScreen?",
      "Would You Like To Run In Fullscreen Mode?",
      QMessageBox::Yes,
      QMessageBox::No | QMessageBox::Default ) )
  {
  case QMessageBox::Yes:
    fs = true;
    break;
  case QMessageBox::No:
    fs = false;
    break;
```

```
  }
```

这里弹出一个消息对话框，让用户选择是否使用全屏模式。

```
  NeHeWidget w( 0, 0, fs );
```

创建一个 NeHeWidget 对象。

```
  a.setMainWidget( &w );
```

设置应用程序的主窗口部件为 w。

```
  w.show();
```

显示 w。

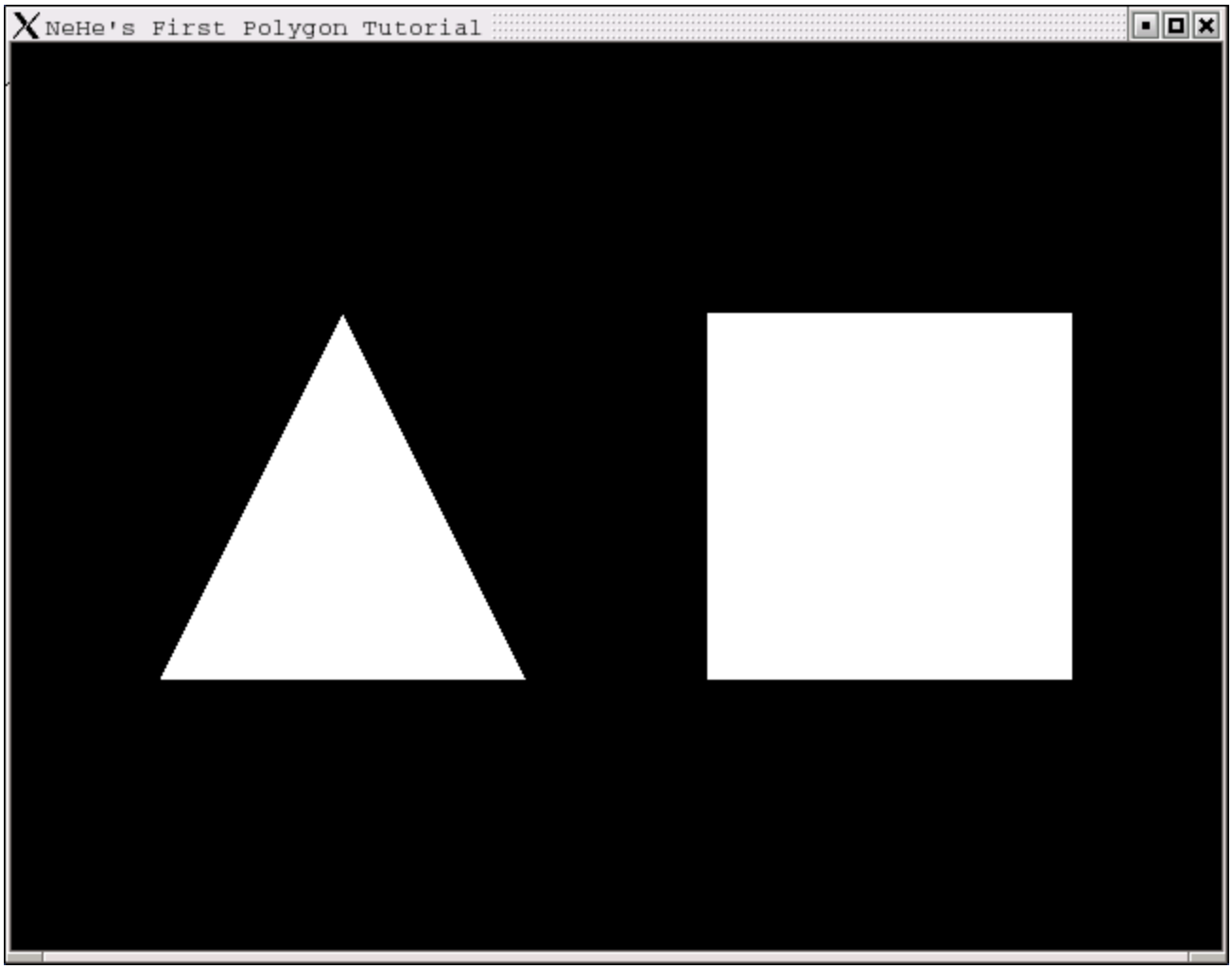
```
  return a.exec();
```

程序返回。

```
}
```

本课程的源代码。

你的第一个多边形



上一课中，我教您如何创建一个 OpenGL 窗口。这一课中，我将教您如何创建三角形和四边形。我们讲使用 GL_TRIANGLES 来创建一个三角形，GL_QUADS 来创建一个四边形。

我们只要修改第一课中的 NeHeWidget 类中的 paintGL()函数就可以了。

NeHeWidget 类

（由 nehewidget.cpp 展开。）

```
void NeHeWidget::paintGL()
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

清除屏幕和深度缓存。

```
  GlLoadIdentity ();
```

重置当前的模型观察矩阵。

当您调用 glLoadIdentity()之后，您实际上将当前点移到了屏幕中心，X 坐标轴从左至右，Y 坐标轴从下至上，Z 坐标轴从里至外。OpenGL 屏幕中心的坐标值是 X 和 Y 轴上的 0.0 点。中心左面的坐标值是负值，右面是正值。移向屏幕顶端是正值，移向屏幕底端是负值。移入屏幕深处是负值，移出屏幕则是正值。

```
glTranslatef( -1.5,  0.0, -6.0 );
```

glTranslatef(x, y, z)沿着 X, Y 和 Z 轴移动。根据前面的次序，下面的代码沿着 X 轴左移 1.5 个单位，Y 轴不动(0.0)，最后移入屏幕 6.0 个单位。注意在 glTranslatef(x, y, z)中当您移动的时候，您并不是相对屏幕中心移动，而是相对与当前所在的屏幕位置。

现在我们已经移到了屏幕的左半部分，并且将视图推入屏幕背后足够的距离以便我们可以看见全部的场景－创建三角形。

```
glBegin( GL_TRIANGLES );
```

开始绘制三角形。

glBegin(GL_TRIANGLES)的意思是开始绘制三角形，glEnd() 告诉 OpenGL 三角形已经创建好了。通常您会需要画 3 个顶点，可以使用 GL_TRIANGLES。在绝大多数的显卡上，绘制三角形是相当快速的。如果要画四个顶点，使用 GL_QUADS 的话会更方便。但据我所知，绝大多数的显卡都使用三角形来为对象着色。最后，如果您想要画更多的顶点时，可以使用 GL_POLYGON。

本节的简单示例中，我们只画一个三角形。如果要画第二个三角形的话，可以在这三点之后，再加三行代码(3 点)。所有六点代码都应包含在 glBegin(GL_TRIANGLES)和 glEnd()之间。在他们之间再不会有多余的点出现，也就是说，(GL_TRIANGLES)和 glEnd()之间的点都是以三点为一个集合的。这同样适用于四边形。如果您知道实在绘制四边形的话，您必须在第一个四点之后，再加上四点为一个集合的点组。另一方面，多边形可以由任意个顶点，(GL_POLYGON)不在乎 glBegin(GL_TRIANGLES)和 glEnd()之间有多少行代码。

```
    glVertex3f(  0.0,  1.0,  0.0 );
```

上顶点。

glBegin 之后的第一行设置了多边形的第一个顶点，glVertex 的第一个参数是 X 坐标，然后依次是 Y 坐标和 Z 坐标。第一个点是上顶点，然后是左下顶点和右下顶点。glEnd()告诉 OpenGL 没有其他点了。这样将显示一个填充的三角形。

CKer 注：这里要注意的是存在两种不同的坐标变换方式，glTranslatef(x, y, z)中的 x, y, z 是相对与您当前所在点的位移，但 glVertex(x,y,z)是相对于 glTranslatef(x,

y, z)移动后的新原点的位移。因而这里可以认为 glTranslate 移动的是坐标原点，glVertex 中的点是相对最新的坐标原点的坐标值。

```
    glVertex3f( -1.0, -1.0,  0.0 );
```

左下顶点。

```
    glVertex3f(  1.0, -1.0,  0.0 );
```

右下顶点。

```
  glEnd();
```

三角形绘制结束。

```
  glTranslatef(  3.0,  0.0,  0.0 );
```

在屏幕的左半部分画完三角形后，我们要移到右半部分来画正方形。为此要再次使用 glTranslate。这次右移，所以 X 坐标值为正值。因为前面左移了 1.5 个单位，这次要先向右移回屏幕中心(1.5 个单位)，再向右移动 1.5 个单位。总共要向右移 3.0 个单位。

```
  glBegin( GL_QUADS );
```

开始绘制四边形。

```
    glVertex3f( -1.0,  1.0,  0.0 );
```

左上顶点。

```
    glVertex3f(  1.0,  1.0,  0.0 );
```

右上顶点。

```
    glVertex3f(  1.0, -1.0,  0.0 );
```

右下顶点。

```
    glVertex3f( -1.0, -1.0,  0.0 );
```

左下顶点。

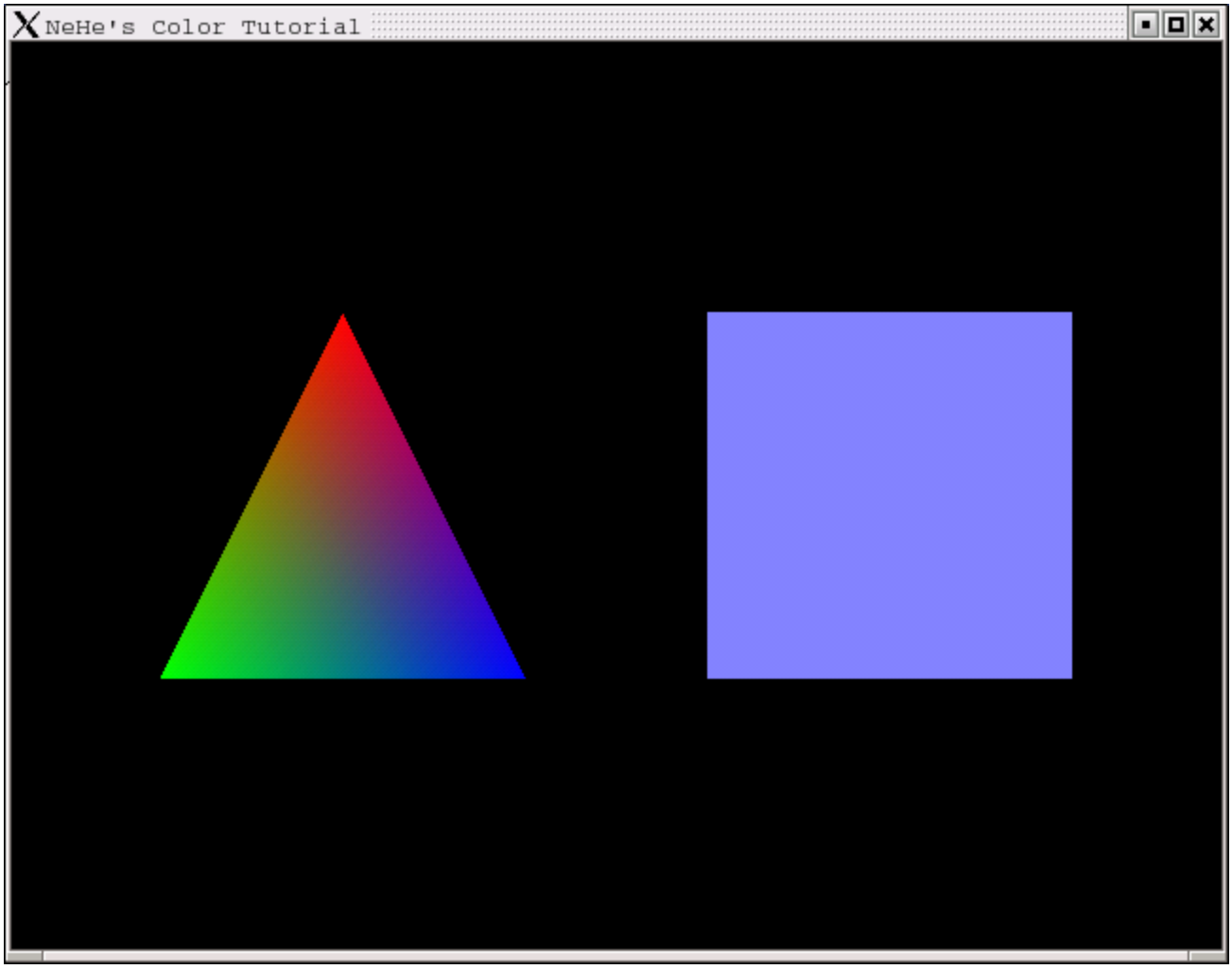
```
  glEnd();
```

四边形绘制结束。

}

本课程的源代码。

上色



上一课中我教给您三角形和四边形的绘制方法。这一课我将教您给三角形和四边形添加两种不同类型的着色方法。使用单调着色（Flat coloring）给四边形涂上固定的一种颜色。使用平滑着色（Smooth coloring）将三角形的三个顶点的不同颜色混合在一起，创建漂亮的色彩混合。

我们只要修改第二课中的 NeHeWidget 类中的 paintGL()函数就可以了。

NeHeWidget 类

（由 nehewidget.cpp 展开。）

```
void NeHeWidget::paintGL()
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glLoadIdentity();

  glTranslatef( -1.5,  0.0, -6.0 );

  glBegin( GL_TRIANGLES );
  glColor3f( 1.0, 0.0, 0.0 );
```

红色。

如果您还记得上节课的内容，这段代码在屏幕的左半部分绘制三角形。这一行代码是我们第一次使用命令 glColor3f(r, g, b)。括号中的三个参数依次是红、绿、蓝三色分量。取值范围可以从 0.0 到 1.0。类似于以前所讲的清除屏幕背景命令。

我们将颜色设为红色（纯红色，无绿色，无蓝色）。

```
    glVertex3f(  0.0,  1.0,  0.0 );
```

上顶点。

接下来的一行代码设置三角形的第一个顶点（三角形的上顶点），并使用当前颜色（红色）来绘制。从现在开始所有的绘制的对象的颜色都是红色，直到我们将红色改变成别的什么颜色。

```
    glColor3f( 0.0, 1.0, 0.0 );
```

绿色。

```
    glVertex3f( -1.0, -1.0,  0.0 );
```

左下顶点。

```
    glColor3f( 0.0, 0.0, 1.0 );
```

蓝色。

```
    glVertex3f(  1.0, -1.0,  0.0 );
```

右下顶点。

```
  glEnd();
```

glEnd()出现后，三角形将被填充。但是因为每个顶点有不同的颜色，因此看起来颜色从每个角喷出，并刚好在三角形的中心汇合，三种颜色相互混合。这就是平滑着色。

```
  glTranslatef(  3.0,  0.0,  0.0 );

  glColor3f( 0.5, 0.5, 1.0 );
```

一次性将颜色设置为蓝色。

现在我们绘制一个单调着色——蓝色的正方形。最重要的是要记住，设置当前色之后绘制的所有东东都是当前色的。以后您所创建的每个工程都要使用颜色。即便是在完全采用纹理贴图的时候，glColor3f 仍旧可以用来调节纹理的色调。等等...，以后再说吧。

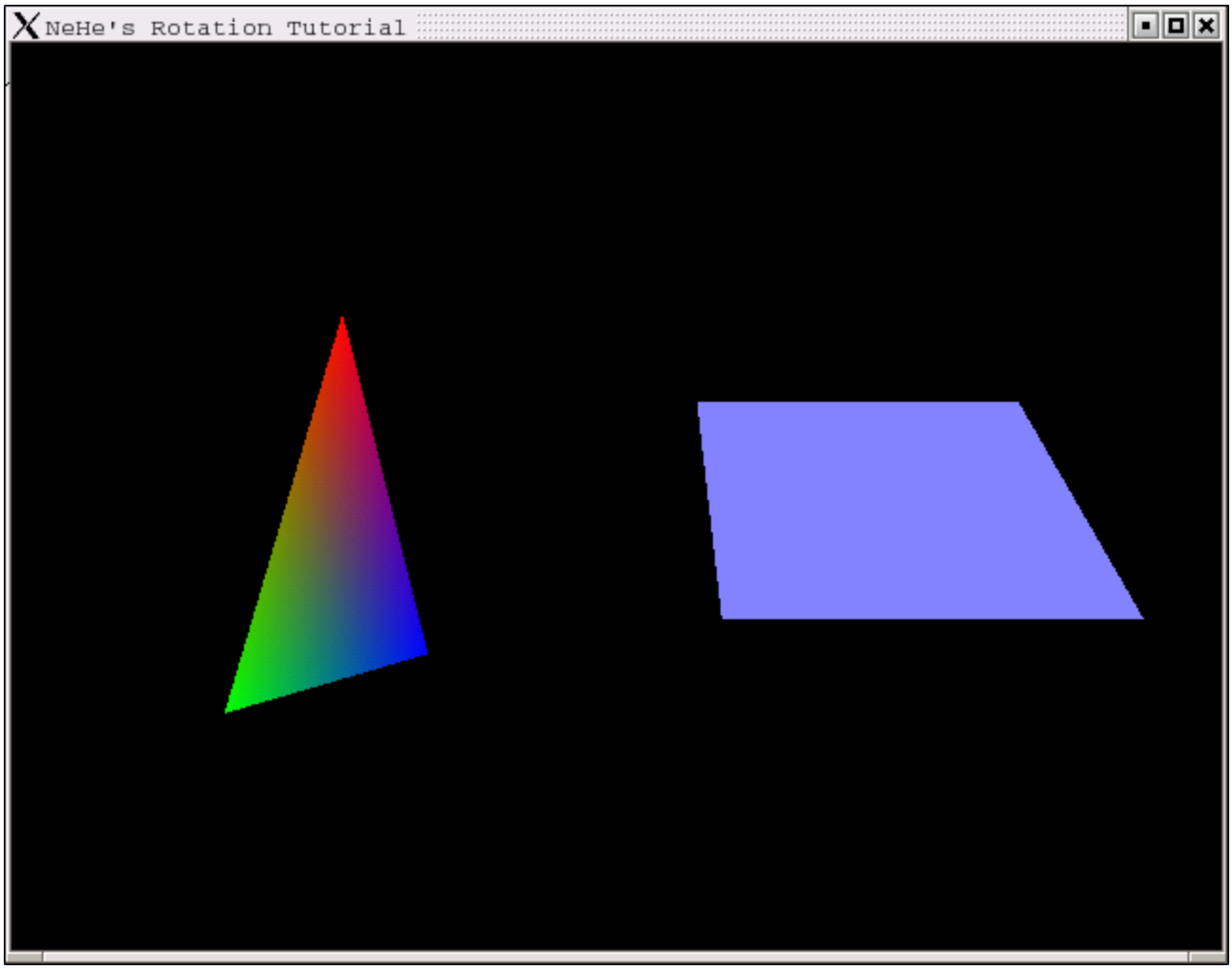
我们必须要做的事只需将颜色一次性的设为我们想采用的颜色(本例采用蓝色)，然后绘制场景。每个顶点都是蓝色的，因为我们没有告诉 OpenGL 要改变顶点的颜色。最后的结果是.....全蓝色的正方形。再说一遍，顺时针绘制的正方形意味着我们所看见的是四边形的背面。

```
  glBegin( GL_QUADS );
    glVertex3f( -1.0,  1.0,  0.0 );
    glVertex3f(  1.0,  1.0,  0.0 );
    glVertex3f(  1.0, -1.0,  0.0 );
    glVertex3f( -1.0, -1.0,  0.0 );
  glEnd();
}
```

在这一课中，我试着尽量详细的解释如何为您的 OpenGL 多边形添加单调和平滑的着色效果的步骤。改改代码中的红绿蓝分量值，看看最后有什么样的结果。

本课程的源代码。

旋转



上一课中我教给您三角形和四边形的着色。这一课我将教您如何将这些彩色对象绕着坐标轴旋转。

其实只需在上节课的代码上增加几行就可以了。

我们将在 NeHeWidget 类中增加两个变量来控制这两个对象的旋转。它们是浮点类型的变量，使得我们能够非常精确地旋转对象。浮点数包含小数位置，这意味着我们无需使用 1、2、3...的角度。你会发现浮点数是 OpenGL 编程的基础。新变量中叫做 rTri 的用来旋转三角形，rQuad 旋转四边形。

NeHeWidget 类

（由 nehewidget.h 展开。）

```
protected:

  bool fullscreen;
  GLfloat rTri;
```

```
  GLfloat rQuad;

};
```

上面就是添加的两个变量。rTri 是用于三角形的角度，rQuad 是用于四边形的角度。

（由 nehewidget.cpp 展开。）

```
NeHeWidget::NeHeWidget( QWidget* parent, const char* name, bool fs )
    : QGLWidget( parent, name )
{
  rTri = 0.0;
  rQuad = 0.0;
  fullscreen = fs;
  setGeometry( 0, 0, 640, 480 );
  setCaption( "NeHe's Rotation Tutorial" );

  if ( fullscreen )
    showFullScreen();
}
```

我们需要在构造函数中给 rTri 和 rQuad 赋初值，都是 0.0。

```
void NeHeWidget::paintGL()
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glLoadIdentity();
  glTranslatef( -1.5,  0.0, -6.0 );
  glRotatef( rTri,  0.0,  1.0,  0.0 );
```

glRotatef(Angle, Xvector, Yvector, Zvector)负责让对象绕某个轴旋转。这个函数有很多用处。 Angle 通常是个变量代表对象转过的角度。Xvector，Yvector 和 Zvector 三个参数则共同决定旋转轴的方向。比如(1, 0, 0)所描述的矢量经过 X 坐标轴的 1 个单位处并且方向向右。(-1, 0, 0)所描述的矢量经过 X 坐标轴的 1 个单位处，但方向向左。

D. Michael Traub：提供了对 Xvector , Yvector 和 Zvector 的上述解释。

为了更好的理解 X, Y 和 Z 的旋转，我举些例子...

X 轴－您正在使用一台台锯。锯片中心的轴从左至右摆放（就像 OpenGL 中的 X 轴）。尖利的锯齿绕着 X 轴狂转，看起来要么向上转，要么向下转。取决于锯片开始转时的方向。这与我们在 OpenGL 中绕着 X 轴旋转什么的情形是一样的。（CKer 注：这会儿您要把脸蛋凑向显示器的话，保准被锯开了花 ^-^。）

Y 轴－假设您正处于一个巨大的龙卷风中心，龙卷风的中心从地面指向天空（就像 OpenGL 中的 Y 轴）。垃圾和碎片围着 Y 轴从左向右或是从右向左狂转不止。这与我们在 OpenGL 中绕着 Y 轴旋转什么的情形是一样的。

Z 轴－您从正前方看着一台风扇。风扇的中心正好朝着您(就像 OpenGL 中的 Z 轴)。风扇的叶片绕着 Z 轴顺时针或逆时针狂转。这与我们在 OpenGL 中绕着 Z 轴旋转什么的情形是一样的。

上面的一行代码中，如果 rtri 等于 7，我们将三角形绕着 Y 轴从左向右旋转 7 。您也可以改变参数的值，让三角形绕着 X 和 Y 轴同时旋转。

```
glBegin( GL_TRIANGLES );
  glColor3f( 1.0, 0.0, 0.0 );
  glVertex3f(  0.0,  1.0,  0.0 );
  glColor3f( 0.0, 1.0, 0.0 );
  glVertex3f( -1.0, -1.0,  0.0 );
  glColor3f( 0.0, 0.0, 1.0 );
  glVertex3f(  1.0, -1.0,  0.0 );
glEnd();
```

上面的绘制三角形的代码没有改变。在屏幕的左面画了一个彩色渐变三角形，并绕着 Y 轴从左向右旋转。

```
glLoadIdentity();
```

我们增加了另一个 glLoadIdentity()调用。目的是为了重置模型观察矩阵。如果我们没有重置，直接调用 glTranslate 的话，会出现意料之外的结果。因为坐标轴已经旋转了，很可能没有朝着您所希望的方向。所以我们本来想要左右移动对象的，就可能变成上下移动了，取决于您将坐标轴旋转了多少角度。试试将 glLoadIdentity() 注释掉之后，会出现什么结果。

重置模型观察矩阵之后，X、Y、Z 轴都以复位，我们调用 glTranslate。您会注意到这次我们只向右移了 1.5 单位，而不是上节课的 3.0 单位。因为我们重置场景的时候，焦点又回到了场景的中心(0.0)处。这样就只需向右移 1.5 单位就够了。当我们移到新位置后，绕 X 轴旋转四边形。正方形将上下转动。

```
glTranslatef(  1.5,  0.0, -6.0 );
glRotatef( rQuad,  1.0,  0.0,  0.0 );
```

绕 X 轴旋转四边形。

```
glColor3f( 0.5, 0.5, 1.0 );
glBegin( GL_QUADS );
  glVertex3f( -1.0,  1.0,  0.0 );
  glVertex3f(  1.0,  1.0,  0.0 );
```

```
    glVertex3f(  1.0, -1.0,  0.0 );
    glVertex3f( -1.0, -1.0,  0.0 );
  glEnd();

  rTri += 0.2;
  rQuad -= 0.15;
```

我们在构造函数中已经将 rTri 和 rQuad 的值设为 0.0，在这里我们每绘制完一次图像，就修改一下这两个变量。两个变量的变化会使对象的旋转角度发生变化。

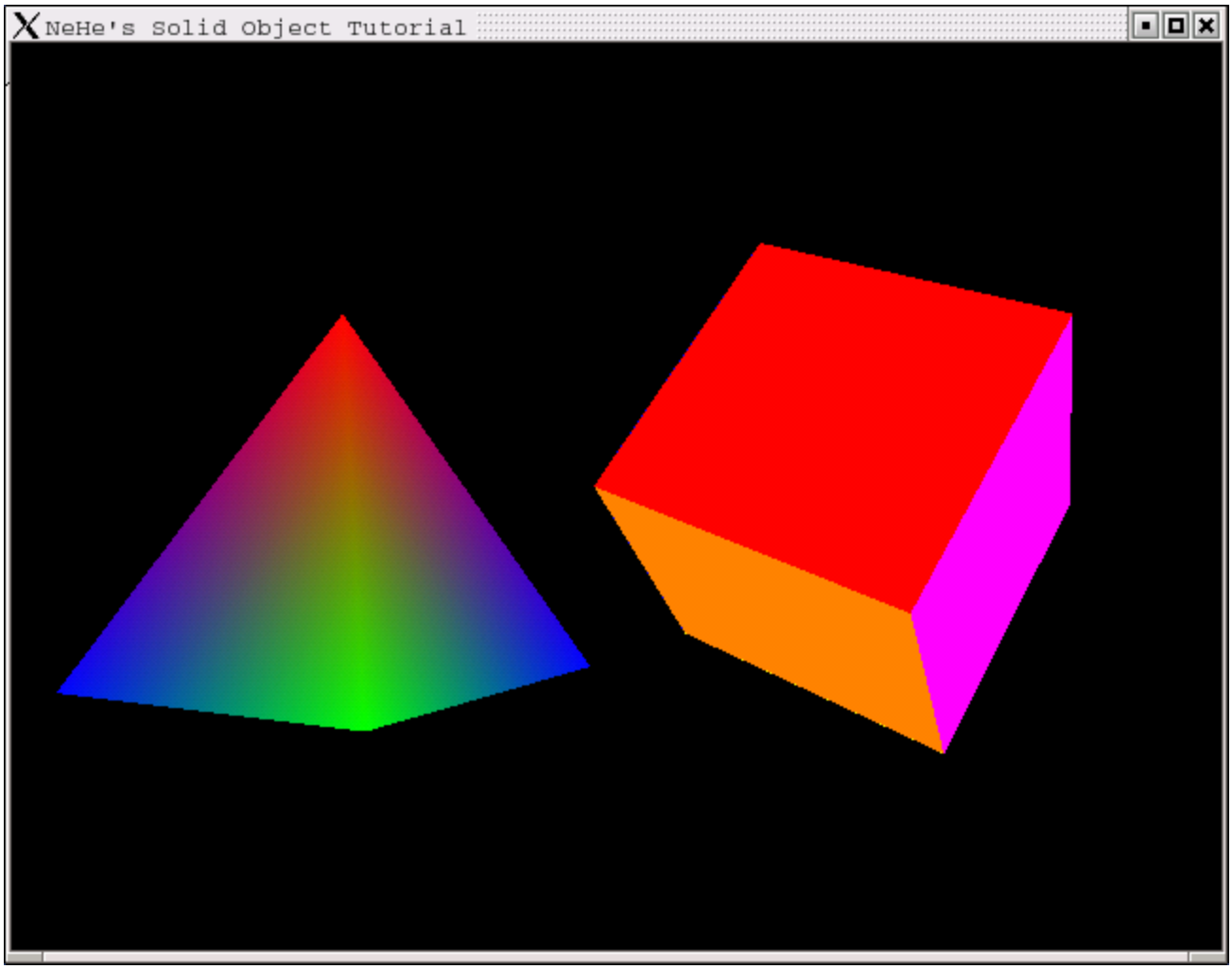
尝试改变下面代码中的+和-，来体会对象旋转的方向是如何改变的。并试着将 0.2 改成 1.0。这个数字越大，物体就转的越快，这个数字越小，物体转的就越慢。

```
}
```

在这一课中，我试着尽量详细的解释如何让对象绕某个轴转动。改改代码，试着让对象绕着 Z 轴、X+Y 轴或者所有三个轴来转动:)。

本课程的源代码。

向三维进军



在上一课的内容上作些扩展，我们现在开始生成真正的三维对象，而不是象前两节课中那样在三维世界中的二维对象。我们给三角形增加一个左侧面，一个右侧面，一个后侧面来生成一个金字塔（四棱锥）。给正方形增加左、右、上、下及背面生成一个立方体。

我们混合金字塔上的颜色，创建一个平滑着色的对象。给立方体的每一面则来个不同的颜色。

其实只需在上节课的代码上增加几行就可以了。

NeHeWidget 类

（由 nehewidget.cpp 展开。）

```
void NeHeWidget::paintGL()
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glLoadIdentity();
```

```
glTranslatef( -1.5,  0.0, -6.0 );
glRotatef( rTri,  0.0,  1.0,  0.0 );
```

有些人可能早已在上节课中的代码上尝试自行创建 3D 对象了。但经常有人来信问我："我的对象怎么不会绕着其自身的轴旋转？看起来总是在满屏乱转。"要让您的对象绕自身的轴旋转，您必须让对象的中心坐标总是(0.0, 0,0, 0,0)。

下面的代码创建一个绕者其中心轴旋转的金字塔。金字塔的上顶点离中心一个单位，底面离中心也是一个单位。上顶点在底面的投影位于底面的中心。

注意所有的面－三角形都是逆时针次序绘制的。这点十分重要，在以后的课程中我会作出解释。现在，您只需明白要么都逆时针，要么都顺时针，但永远不要将两种次序混在一起，除非您有足够的理由必须这么做。

```
glBegin( GL_TRIANGLES );
  glColor3f( 1.0, 0.0, 0.0 );
  glVertex3f(  0.0,  1.0,  0.0 );
  glColor3f( 0.0, 1.0, 0.0 );
  glVertex3f( -1.0, -1.0,  1.0 );
  glColor3f( 0.0, 0.0, 1.0 );
  glVertex3f(  1.0, -1.0,  1.0 );
```

上面是我们绘制的金字塔的前侧面。因为所有的面都共享上顶点，我们将这点在所有的三角形中都设置为红色。底边上的两个顶点的颜色则是互斥的。前侧面的左下顶点是绿色的，右下顶点是蓝色的。这样相邻右侧面的左下顶点是蓝色的，右下顶点是绿色的。这样四边形的底面上的点的颜色都是间隔排列的。

还应注意到后面的三个侧面和前侧面处于同一个 glBegin(GL_TRIANGLES)和 glEnd()语句中间。因为我们是通过三角形来构造这个金字塔的。OpenGL 知道每三个点构成一个三角形。当它画完一个三角形之后，如果还有余下的点出现，它就以为新的三角形要开始绘制了。OpenGL 在这里并不会将四点画成一个四边形，而是假定新的三角形开始了。所以千万不要无意中增加任何多余的点。

```
  glColor3f( 1.0, 0.0, 0.0 );
  glVertex3f(  0.0,  1.0,  0.0 );
  glColor3f( 0.0, 0.0, 1.0 );
  glVertex3f(  1.0, -1.0,  1.0 );
  glColor3f( 0.0, 1.0, 0.0 );
  glVertex3f(  1.0, -1.0, -1.0 );
```

绘制右侧面。注意其底边上的两个顶点的 X 坐标位于中心右侧的一个单位处。顶点则位于 Y 轴上的一单位处，且 Z 坐标正好处于底边的两顶点的 Z 坐标中心。右侧面从上顶点开始向外侧倾斜至底边上。

这次的左下顶点用蓝色绘制，以保持与前侧面的右下顶点的一致。蓝色将从这个角向金字塔的前侧面和右侧面扩展并与其他颜色混合。

```
glColor3f( 1.0, 0.0, 0.0 );
glVertex3f(  0.0,  1.0,  0.0 );
glColor3f( 0.0, 1.0, 0.0 );
glVertex3f(  1.0, -1.0, -1.0 );
glColor3f( 0.0, 0.0, 1.0 );
glVertex3f( -1.0, -1.0, -1.0 );
```

后侧面。再次切换颜色。左下顶点又回到绿色，因为后侧面与右侧面共享这个角。

```
glColor3f( 1.0, 0.0, 0.0 );
glVertex3f(  0.0,  1.0,  0.0 );
glColor3f( 0.0, 0.0, 1.0 );
glVertex3f( -1.0, -1.0, -1.0 );
glColor3f( 0.0, 1.0, 0.0 );
glVertex3f( -1.0, -1.0,  1.0 );
```

最后画左侧面。又要切换颜色。左下顶点是蓝色，与后侧面的右下顶点相同。右下顶点是蓝色，与前侧面的左下顶点相同。

到这里金字塔就画完了。因为金字塔只绕着 Y 轴旋转，我们永远都看不见底面，因而没有必要添加底面。如果您觉得有经验了，尝试增加底面（正方形），并将金字塔绕 X 轴旋转来看看您是否作对了。确保底面四个顶点的颜色与侧面的颜色相匹配。

```
glEnd();
```

接下来开始画立方体。他由六个四边形组成。所有的四边形都以逆时针次序绘制。就是说先画右上角，然后左上角、左下角、最后右下角。您也许认为画立方体的背面的时候这个次序看起来好像顺时针，但别忘了我们从立方体的背后看背面的时候，与您现在所想的正好相反。（译者注：您是从立方体的外面来观察立方体的。)

```
glLoadIdentity();
glTranslatef(  1.5,  0.0, -7.0 );
```

注意到这次我们将立方体移地更远离屏幕了。因为立方体的大小要比金字塔大，同样移入 6 个单位时，立方体看起来要大的多。这是透视的缘故。越远的对象看起来越小 :) 。

```
glRotatef( rQuad,  1.0,  1.0,  1.0 );

glBegin( GL_QUADS );
```

```
glColor3f( 0.0, 1.0, 0.0 );
glVertex3f(  1.0,  1.0, -1.0 );
glVertex3f( -1.0,  1.0, -1.0 );
glVertex3f( -1.0,  1.0,  1.0 );
glVertex3f(  1.0,  1.0,  1.0 );
```

先画立方体的顶面。从中心上移一单位，注意 Y 坐标始终为一单位，表示这个四边形与 Z 轴平行。先画右上顶点，向右一单位，再屏幕向里一单位。然后左上顶点，向左一单位，再屏幕向里一单位。然后是靠近观察者的左下和右下顶点。就是屏幕往外一单位。

```
glColor3f( 1.0, 0.5, 0.0 );
glVertex3f(  1.0, -1.0,  1.0 );
glVertex3f( -1.0, -1.0,  1.0 );
glVertex3f( -1.0, -1.0, -1.0 );
glVertex3f(  1.0, -1.0, -1.0 );
```

底面的画法和顶面十分类似。只是 Y 坐标变成了-1。如果我们从立方体的下面来看立方体的话，您会注意到右上角离观察者最近，因此我们先画离观察者最近的顶点。然后是左上顶点最后才是屏幕里面的左下和右下顶点。

如果您真的不在乎绘制多边形的次序（顺时针或者逆时针）的话，您可以直接拷贝顶面的代码，将 Y 坐标从 1 改成-1，也能够工作。但一旦您进入象纹理映射这样的领域时，忽略绘制次序会导致十分怪异的结果。

```
glColor3f( 1.0, 0.0, 0.0 );
glVertex3f(  1.0,  1.0,  1.0 );
glVertex3f( -1.0,  1.0,  1.0 );
glVertex3f( -1.0, -1.0,  1.0 );
glVertex3f(  1.0, -1.0,  1.0 );
```

立方体的前面。保持 Z 坐标为一单位，前面正对着我们。

```
glColor3f( 1.0, 1.0, 0.0 );
glVertex3f(  1.0, -1.0, -1.0 );
glVertex3f( -1.0, -1.0, -1.0 );
glVertex3f( -1.0,  1.0, -1.0 );
glVertex3f(  1.0,  1.0, -1.0 );
```

立方体后面的绘制方法与前面类似。只是位于屏幕的里面。注意 Z 坐标现在保持-1 不变。

```
glColor3f( 0.0, 0.0, 1.0 );
glVertex3f( -1.0,  1.0,  1.0 );
glVertex3f( -1.0,  1.0, -1.0 );
```

```
    glVertex3f( -1.0, -1.0, -1.0 );
    glVertex3f( -1.0, -1.0,  1.0 );
```

还剩两个面就完成了。您会注意到总有一个坐标保持不变。这一次换成了 X 坐标。因为我们在画左侧面。

```
    glColor3f( 1.0, 0.0, 1.0 );
    glVertex3f(  1.0,  1.0, -1.0 );
    glVertex3f(  1.0,  1.0,  1.0 );
    glVertex3f(  1.0, -1.0,  1.0 );
    glVertex3f(  1.0, -1.0, -1.0 );
```

立方体的最后一个面了。X 坐标保持为一单位。逆时针绘制。您愿意的话，留着这个面不画也可以，这样就是一个盒子:)

```
  glEnd();
```

或者您要是有兴趣可以改变立方体所有顶点的色彩值，象金字塔那样混合颜色。您会看见一个非常漂亮的彩色立方体，各种颜色在它的各个表面流淌。

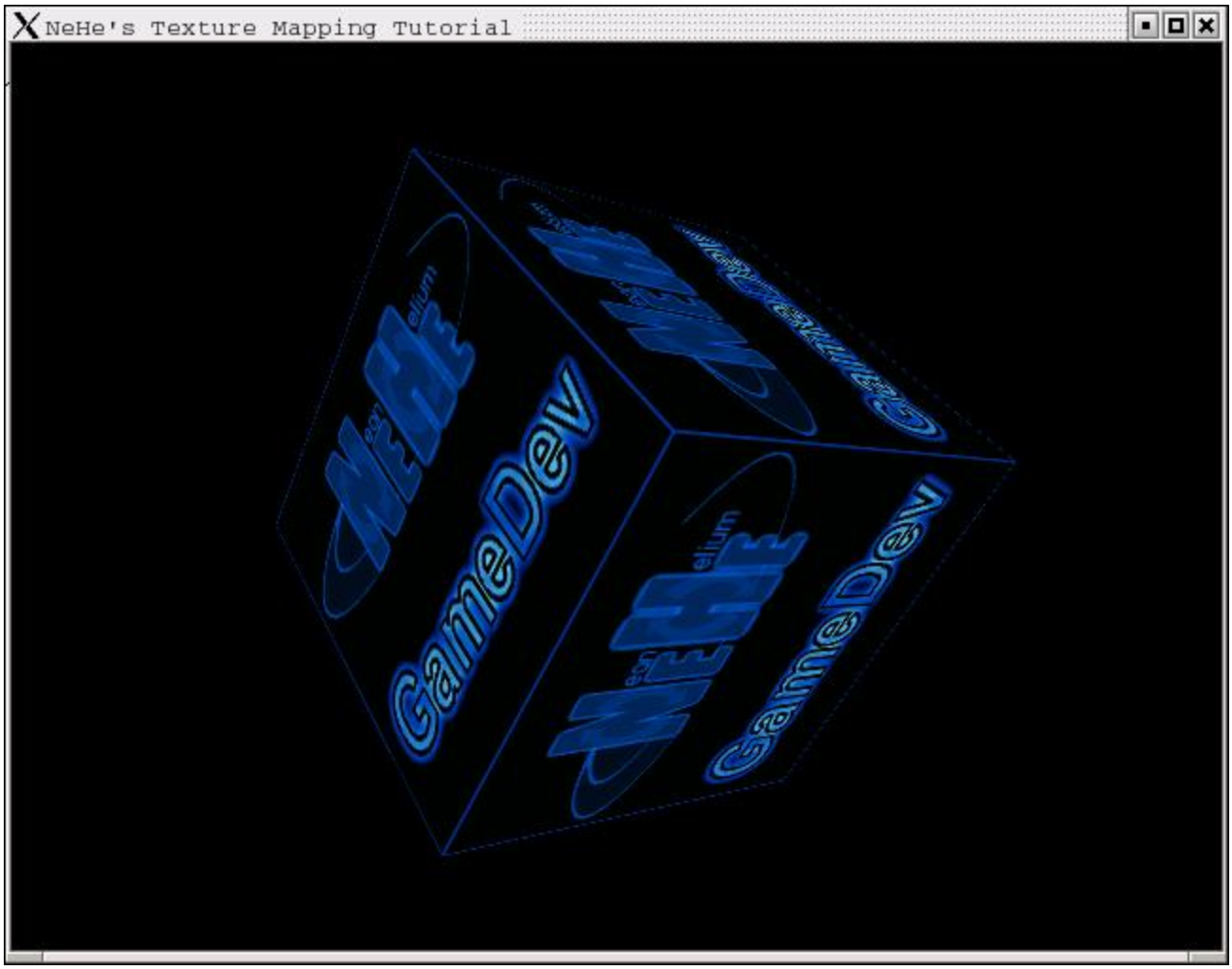
```
  rTri  += 0.2;
  rQuad -= 0.15;
}
```

这一课又结束了。到这里您应该已经较好的掌握了在三维空间创建对象的方法。必须将 OpenGL 屏幕想象成一张很大的画纸，后面还带着许多透明的层。差不多就是个由大量的点组成的立方体。这些点从左至右、从上至下、从前到后的布满了这个立方体。如果您能想象的出在屏幕的深度方向，应该在设计新三维对象时没有任何问题。

如果您对三维空间的理解很困难的话，千万不要灰心！刚开始的时候，领会这些内容会很难。象立方体这样的对象是您练习的好例子。继续努力吧！

本课程的源代码。

纹理映射



学习 texture map 纹理映射（贴图）有很多好处。比方说您想让一颗导弹飞过屏幕。根据前几课的知识，我们最可行的办法可能是很多个多边形来构建导弹的轮廓并加上有趣的颜色。使用纹理映射，您可以使用真实的导弹图像并让它飞过屏幕。您觉得哪个更好看？照片还是一大堆三角形和四边形？使用纹理映射的好处还不止是更好看，而且您的程序运行会更快。导弹贴图可能只是一个飞过窗口的四边形。一个由多边形构建而来的导弹却很可能包括成百上千的多边形。很显然，贴图极大的节省了 CPU 时间。

我们要在第一课的代码上增加几行就可以了。

我们将要增加一个 loadGLTextures()函数来处理有关纹理操作的。我们将在 NeHeWidget 类中增加三个变量 xRot、yRot、zRot 来处理立方体的旋转。还有一个用来存储纹理的 texture[1]。

NeHeWidget 类

（由 nehewidget.h 展开。）

```
protected:

  void loadGLTextures();
```

在这个函数中我们会载入指定的图片并生成相应当纹理。

```
protected:

  bool fullscreen;
  GLfloat xRot, yRot, zRot;
  GLuint texture[1];

};
```

上面就是添加的三个变量 xRot、yRot、zRot 来处理立方体在三个方向上的旋转。texture[1]用来存储一个纹理。

（由 nehewidget.cpp 展开。）

```
NeHeWidget::NeHeWidget( QWidget* parent, const char* name, bool fs )
    : QGLWidget( parent, name )
{
  xRot = yRot = zRot = 0.0;
  fullscreen = fs;
  setGeometry( 0, 0, 640, 480 );
  setCaption( "NeHe's Texture Mapping Tutorial" );

  if ( fullscreen )
    showFullScreen();
}
```

我们需要在构造函数中给 xRot、yRot、zRot 赋初值，都是 0.0。

```
void NeHeWidget::loadGLTextures()
{
  QImage tex, buf;
  if ( !buf.load( "./data/NeHe.bmp" ) )
```

载入纹理图片。这里使用了 QImage 类。

```
  {
    qWarning( "Could not read image file, using single-color instead." );
    QImage dummy( 128, 128, 32 );
    dummy.fill( Qt::green.rgb() );
    buf = dummy;
```

如果载入不成功，自动生成一个 128*128 的 32 位色的绿色图片。

```
}
tex = QGLWidget::convertToGLFormat( buf );
```

这里使用了 QGLWidget 类中提供的一个静态函数 converToGLFormat()，专门用来转换图片的，具体情况请参见相应文档。

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

创建一个纹理。告诉 OpenGL 我们想生成一个纹理名字(如果您想载入多个纹理，加大数字)。值得注意的是，开始我们使用 GLuint texture[1] 来创建一个纹理的存储空间，您也许会认为第一个纹理就是存放在 &texture[1] 中的，但这是错的。正确的地址应该是 &texture[0] 。同样如果使用 GLuint texture[2] 的话，第二个纹理存放在 texture[1] 中。

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

使用来自位图数据生成的典型纹理。告诉 OpenGL 将纹理名字 texture[0]绑定到纹理目标上。2D 纹理只有高度（在 Y 轴上）和宽度（在 X 轴上）。主函数将纹理名字指派给纹理数据。本例中我们告知 OpenGL， &texture[0]处的内存已经可用。我们创建的纹理将存储在&texture[0]的指向的内存区域。

```
glTexImage2D( GL_TEXTURE_2D, 0, 3, tex.width(), tex.height(), 0,
    GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() );
```

这里真正的创建纹理。GL_TEXTURE_2D 告诉 OpenGL 此纹理是一个 2D 纹理。数字零代表图像的详细程度，通常就由它为零去了。数字三是数据的成分数。因为图像是由红色数据，绿色数据，蓝色数据三种组分组成。 tex.width()是纹理的宽度。tex.height()是纹理的高度。数字零是边框的值，一般就是零。GL_RGBA 告诉 OpenGL 图像数据由红、绿、蓝三色数据以及 alpha 通道数据组成，这个是由于 QGLWidget 类的 converToGLFormat()函数的原因。 GL_UNSIGNED_BYTE 意味着组成图像的数据是无符号字节类型的。最后 tex.bits()告诉 OpenGL 纹理数据的来源。

```
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
```

上面的两行告诉 OpenGL 在显示图像时，当它比放大得原始的纹理大（GL_TEXTURE_MAG_FILTER）或缩小得比原始得纹理小（GL_TEXTURE_MIN_FILTER）时 OpenGL 采用的滤波方式。通常这两种情况下我都采用 GL_LINEAR。这使得纹理从很远处到离屏幕很近时都平滑显示。使用 GL_LINEAR 需要 CPU 和显卡做更多的运算。如果您的机器很慢，您也许应该采用 GL_NEAREST。过滤的纹理在放大的时候，看起来斑驳的很。您也可以结合这两种滤波方式。在近处时使用 GL_LINEAR，远处时 GL_NEAREST。

```
}
```

loadGLTextures()函数就是用来载入纹理的。

```
void NeHeWidget::paintGL()
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glLoadIdentity();
  glTranslatef(  0.0,  0.0, -5.0 );

  glRotatef( xRot,  1.0,  0.0,  0.0 );
  glRotatef( yRot,  0.0,  1.0,  0.0 );
  glRotatef( zRot,  0.0,  0.0,  1.0 );
```

根据 xRot、yRot、zRot 的实际值来旋转正方体。

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

选择我们使用的纹理。如果您在您的场景中使用多个纹理，您应该使用来 glBindTexture(GL_TEXTURE_2D, texture[所使用纹理对应的数字]) 选择要绑定的纹理。当您想改变纹理时，应该绑定新的纹理。有一点值得指出的是，您不能在 glBegin()和 glEnd()之间绑定纹理，必须在 glBegin()之前或 glEnd()之后绑定。注意我们在上面是如何使用 glBindTexture 来指定和绑定纹理的。

```
  glBegin( GL_QUADS );
```

为了将纹理正确的映射到四边形上，您必须将纹理的右上角映射到四边形的右上角，纹理的左上角映射到四边形的左上角，纹理的右下角映射到四边形的右下角，纹理的左下角映射到四边形的左下角。如果映射错误的话，图像显示时可能上下颠倒，侧向一边或者什么都不是。

glTexCoord2f 的第一个参数是 X 坐标。0.0 是纹理的左侧。0.5 是纹理的中点，1.0 是纹理的右侧。glTexCoord2f 的第二个参数是 Y 坐标。0.0 是纹理的底部。0.5 是纹理的中点，1.0 是纹理的顶部。

所以纹理的左上坐标是 X：0.0，Y：1.0f，四边形的左上顶点是 X：-1.0，Y：1.0。其余三点依此类推。

试着玩玩 glTexCoord2f 的 X、Y 坐标参数。把 1.0 改为 0.5 将只显示纹理的左半部分，把 0.0 改为 0.5 将只显示纹理的右半部分。

```
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0,  1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0,  1.0 );
```

前面。

```
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0, -1.0 );
```

后面。

```
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0,  1.0,  1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0,  1.0,  1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );
```

顶面。

```
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0, -1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );
```

底面。

```
    glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0,  1.0,  1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );
```

右面。

```
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0,  1.0,  1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
```

左面。

```
  glEnd();

  xRot += 0.3;
  yRot += 0.2;
  zRot += 0.4;
```

现在改变 xRot、yRot、zRot 的值。尝试变化每次各变量的改变值来调节立方体的旋转速度，或改变+/-号来调节立方体的旋转方向。

```
}
void NeHeWidget::initializeGL()
{
  loadGLTextures();
```

载入纹理。

```
  glEnable( GL_TEXTURE_2D );
```

启用纹理。如果没有启用的话，你的对象看起来永远都是纯白色，这一定不是什么好事。

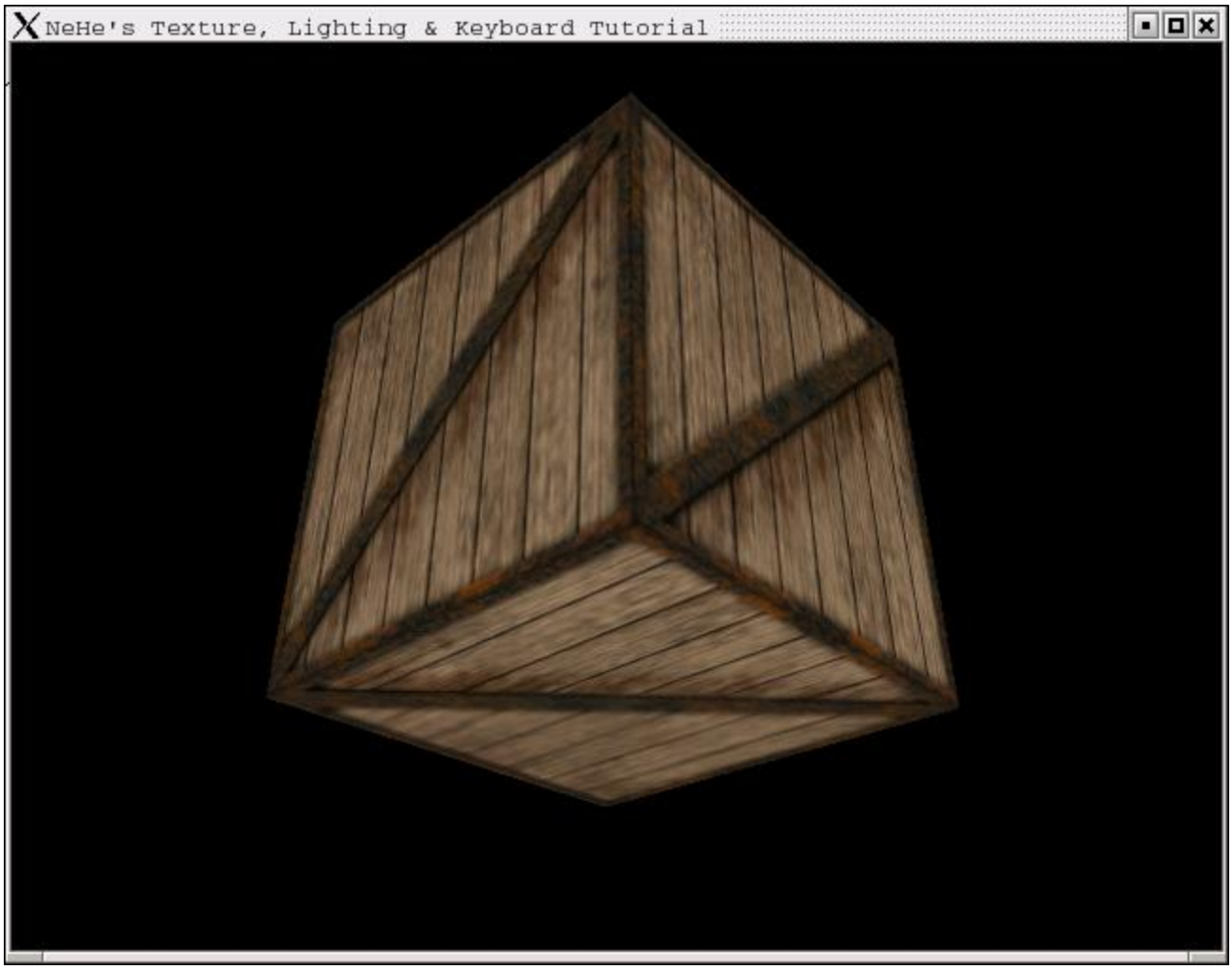
```
  glShadeModel( GL_SMOOTH );
  glClearColor( 0.0, 0.0, 0.0, 0.5 );
  glClearDepth( 1.0 );
  glEnable( GL_DEPTH_TEST );
  glDepthFunc( GL_LEQUAL );
  glHint( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
}
```

现在您应该比较好的理解纹理映射（贴图）了。您应该掌握了给任意四边形表面贴上您所喜爱的图像的技术。一旦您对 2D 纹理映射的理解感到自信的时候，试试给立方体的六个面贴上不同的纹理。

当您理解纹理坐标的概念后，纹理映射并不难理解。

本课程的源代码。

纹理滤波、光源和键盘控制



这一课我会教您如何使用三种不同的纹理滤波方式。教您如何使用键盘来移动场景中的对象，还会教您在 OpenGL 场景中应用简单的光照。这一课包含了很多内容，如果您对前面的课程有疑问的话，先回头复习一下。进入后面的代码之前，很好的理解基础知识十分重要。

我们要在第一课的代码上进行改动就可以了。

我们将要增加一个 loadGLTextures()函数来处理有关纹理操作的。我们将增加一些变量，稍后我们对这些变量进行解释。

NeHeWidget 类

（由 nehewidget.h 展开。）

```
protected:

  void loadGLTextures();
```

在这个函数中我们会载入指定的图片并生成相应当纹理。

```
protected:

  bool fullscreen;
  GLfloat xRot, yRot, zRot;
  GLfloat zoom;
  GLfloat xSpeed, ySpeed;
  GLuint texture[3];
  GLuint filter;

  bool light;

};
```

上面就是添加的三个变量 xRot、yRot、zRot 来处理立方体在三个方向上的旋转。zoom 是场景深入屏幕的距离。xSpeed 和 ySpeed 是立方体在 X 轴和 Y 轴上旋转的速度。texture[3]用来存储三个纹理。filter 表明的是使用哪个纹理。light 是说明现在是否使用光源。

（由 nehewidget.cpp 展开。）

```
GLfloat lightAmbient[4] = { 0.5, 0.5, 0.5, 1.0 };
GLfloat lightDiffuse[4] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat lightPosition[4] = { 0.0, 0.0, 2.0, 1.0 };
```

这里定义了三个数组，它们描述的是和光源有关的信息。

我们将使用两种不同的光。第一种称为环境光。环境光来自于四面八方。所有场景中的对象都处于环境光的照射中。第二种类型的光源叫做漫射光。漫射光由特定的光源产生，并在您的场景中的对象表面上产生反射。处于漫射光直接照射下的任何对象表面都变得很亮，而几乎未被照射到的区域就显得要暗一些。这样在我们所创建的木板箱的棱边上就会产生的很不错的阴影效果。

创建光源的过程和颜色的创建完全一致。前三个参数分别是 RGB 三色分量，最后一个是 alpha 通道参数。

因此，第一行有关 lightAmbient 的代码使我们得到的是半亮（0.5）的白色环境光。如果没有环境光，未被漫射光照到的地方会变得十分黑暗。

第二行有关 lightDiffuse 的代码使我们生成最亮的漫射光。所有的参数值都取成最大值 1.0。它将照在我们木板箱的前面，看起来挺好。

第三行有关 lightPosition 的代码使我们保存光源的位置。前三个参数和 glTranslate 中的一样。依次分别是 XYZ 轴上的位移。由于我们想要光线直接照射在木箱的

正面，所以 XY 轴上的位移都是 0.0。第三个值是 Z 轴上的位移。为了保证光线总在木箱的前面，所以我们将光源的位置朝着观察者（就是您哪。）挪出屏幕。我们通常将屏幕也就是显示器的屏幕玻璃所处的位置称作 Z 轴的 0.0 点。所以 Z 轴上的位移最后定为 2.0。假如您能够看见光源的话，它就浮在您显示器的前方。当然，如果木箱不在显示器的屏幕玻璃后面的话，您也无法看见箱子。最后一个参数取为 1.0。这将告诉 OpenGL 这里指定的坐标就是光源的位置，以后的教程中我会多加解释。

```
NeHeWidget::NeHeWidget( QWidget* parent, const char* name, bool fs )
    : QGLWidget( parent, name )
{
  xRot = yRot = zRot = 0.0;
  zoom = -5.0;
  xSpeed = ySpeed = 0.0;

  filter = 0;

  light = false;

  fullscreen = fs;
  setGeometry( 0, 0, 640, 480 );
  setCaption( "NeHe's Texture, Lighting & Keyboard Tutorial" );

  if ( fullscreen )
    showFullScreen();
}
```

我们需要在构造函数中给各个变量赋初值。xRot、yRot、zRot 是 0.0。zoom 是-5.0。xSpeed 和 ySpeed 都是 0。filter 是 0。light 是 false。

```
void NeHeWidget::loadGLTextures()
{
  QImage tex, buf;
  if ( !buf.load( "./data/Crate.bmp" ) )
  {
    qWarning("Could not read image file, using single-color instead.");
    QImage dummy( 128, 128, 32 );
    dummy.fill( Qt::green.rgb() );
    buf = dummy;
  }
  tex = QGLWidget::convertToGLFormat( buf );

  glGenTextures( 3, &texture[0] );
```

这一部分，上一章讲过了。我们这里创建了 3 个纹理。

```
  glBindTexture( GL_TEXTURE_2D, texture[0] );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
  glTexImage2D( GL_TEXTURE_2D, 0, 3, tex.width(), tex.height(), 0,
      GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() );
```

第六课中我们使用了线性滤波的纹理贴图。这需要机器有相当高的处理能力，但它们看起来很不错。这一课中，我们接着要创建的第一种纹理使用 GL_NEAREST 方式。从原理上讲，这种方式没有真正进行滤波。它只占用很小的处理能力，看起来也很差。唯一的好处是这样我们的工程在很快和很慢的机器上都可以正常运行。您会注意到我们在 MIN 和 MAG 时都采用了 GL_NEAREST,你可以混合使用 GL_NEAREST 和 GL_LINEAR。纹理看起来效果会好些，但我们更关心速度，所以全采用低质量贴图。MIN_FILTER 在图像绘制时小于贴图的原始尺寸时采用。MAG_FILTER 在图像绘制时大于贴图的原始尺寸时采用。

```
  glBindTexture( GL_TEXTURE_2D, texture[1] );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
  glTexImage2D( GL_TEXTURE_2D, 0, 3, tex.width(), tex.height(), 0,
      GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() );
```

这个纹理与第六课的相同，线性滤波。唯一的不同是这次放在了 texture[1]中。因为这是第二个纹理。如果放在 texture[0]中的话，它将覆盖前面创建的 GL_NEAREST 纹理。

```
  glBindTexture( GL_TEXTURE_2D, texture[2] );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST );
```

这里是创建纹理的新方法。Mipmapping！您可能会注意到当图像在屏幕上变得很小的时候，很多细节将会丢失。刚才还很不错的图案变得很难看。当您告诉 OpenGL 创建一个 mipmapped 的纹理后，OpenGL 将尝试创建不同尺寸的高质量纹理。当您向屏幕绘制一个 mipmapped 纹理的时候，OpenGL 将选择它已经创建的外观最佳的纹理(带有更多细节)来绘制，而不仅仅是缩放原先的图像（这将导致细节丢失）。

我曾经说过有办法可以绕过 OpenGL 对纹理宽度和高度所加的限制——64、128、256，等等。办法就是 gluBuild2DMipmaps。据我的发现，您可以使用任意的位图来创建纹理。OpenGL 将自动将它缩放到正常的大小。

因为是第三个纹理，我们将它存到 texture[2]。这样本课中的三个纹理全都创建好了。 gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, tex.width(), tex.height(), GL_RGBA, GL_UNSIGNED_BYTE, tex.bits());

这一行生成 mipmapped 纹理。我们使用三种颜色（红，绿，蓝）来生成一个 2D 纹理。tex.width()是位图宽度，tex.height()是位图高度，extureImage[0]->sizeY 是位图高度，GL_RGBA 意味着我们依次使用 RGBA 色彩。GL_UNSIGNED_BYTE 意味着纹理数据的单位是字节。tex.bits()指向我们创建纹理所用的位图。

}

loadGLTextures()函数就是用来载入纹理的。

```
void NeHeWidget::paintGL()
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glLoadIdentity();
  glTranslatef(  0.0,  0.0, zoom );

  glRotatef( xRot,  1.0,  0.0,  0.0 );
  glRotatef( yRot,  0.0,  1.0,  0.0 );

  glBindTexture( GL_TEXTURE_2D, texture[filter] );
```

根据 filter 变量来决定使用哪个纹理。

```
  glBegin( GL_QUADS );
    glNormal3f( 0.0, 0.0, 1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0,  1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0,  1.0 );

    glNormal3f( 0.0, 0.0, -1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0, -1.0 );

    glNormal3f( 0.0, 1.0, 0.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0,  1.0,  1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0,  1.0,  1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );

    glNormal3f( 0.0, -1.0, 0.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0, -1.0, -1.0 );
```

```
    glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );

    glNormal3f( 1.0, 0.0, 0.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0,  1.0,  1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );

    glNormal3f( -1.0, 0.0, 0.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0,  1.0,  1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
  glEnd();
```

这里绘制正方体的方法，上一章已经讲解过了。

```
  xRot += xSpeed;
  yRot += ySpeed;
```

将 xRot 和 yRot 的旋转值分别增加 xSpeed 和 ySpeed 个单位。xSpeed 和 ySpeed 的值越大，立方体转得就越快。

```
}
void NeHeWidget::initializeGL()
{
  loadGLTextures();

  glEnable( GL_TEXTURE_2D );
  glShadeModel( GL_SMOOTH );
  glClearColor( 0.0, 0.0, 0.0, 0.5 );
  glClearDepth( 1.0 );
  glEnable( GL_DEPTH_TEST );
  glDepthFunc( GL_LEQUAL );
  glHint( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );

  glLightfv( GL_LIGHT1, GL_AMBIENT, lightAmbient );
  glLightfv( GL_LIGHT1, GL_DIFFUSE, lightDiffuse );
  glLightfv( GL_LIGHT1, GL_POSITION, lightPosition );

  glEnable( GL_LIGHT1 );
```

这里开始设置光源。第一行设置环境光的发光量，光源 GL_LIGHT1 开始发光。这一课的开始处我们我们将环境光的发光量存放在 lightAmbient 数组中。现在我们就使用此数组（半亮度环境光）。

接下来我们设置漫射光的发光量。它存放在 lightDiffuse 数组中（全亮度白光）。

然后设置光源的位置。位置存放在 lightPosition 数组中（正好位于木箱前面的中心，X－0.0，Y－0.0，Z 方向移向观察者 2 个单位，位于屏幕外面）。

最后，我们启用一号光源。我们还没有启用 GL_LIGHTING，所以您看不见任何光线。记住：只对光源进行设置、定位、甚至启用，光源都不会工作。除非我们启用 GL_LIGHTING。

```
}
void NeHeWidget::keyPressEvent( QKeyEvent *e )
{
  switch ( e->key() )
  {
  case Qt::Key_L:
    light = !light;
    if ( !light )
    {
      glDisable( GL_LIGHTING );
    }
    else
    {
      glEnable( GL_LIGHTING );
    }
    updateGL();
    break;
```

按下了 L 键，就可以切换是否打开光源。

```
  case Qt::Key_F:
    filter += 1;;
    if ( filter > 2 )
    {
      filter = 0;
    }
    updateGL();
    break;
```

按下了 F 键，就可以转换一下所使用的纹理（就是变换了纹理滤波方式的纹理）。

```
  case Qt::Key_Prior:
```

```
    zoom -= 0.2;
    updateGL();
    break;
```

按下了 PageUp 键，将木箱移向屏幕内部。

```
  case Qt::Key_Next:
    zoom += 0.2;
    updateGL();
    break;
```

按下了 PageDown 键，将木箱移向屏幕外部。

```
  case Qt::Key_Up:
    xSpeed -= 0.01;
    updateGL();
    break;
```

按下了 Up 方向键，减少 xSpeed。

```
  case Qt::Key_Down:
    xSpeed += 0.01;
    updateGL();
    break;
```

按下了 Dowm 方向键，增加 xSpeed。

```
  case Qt::Key_Right:
    ySpeed += 0.01;
    updateGL();
    break;
```

按下了 Right 方向键，增加 ySpeed。

```
  case Qt::Key_Left:
    ySpeed -= 0.01;
    updateGL();
    break;
```

按下了 Left 方向键，减少 ySpeed。

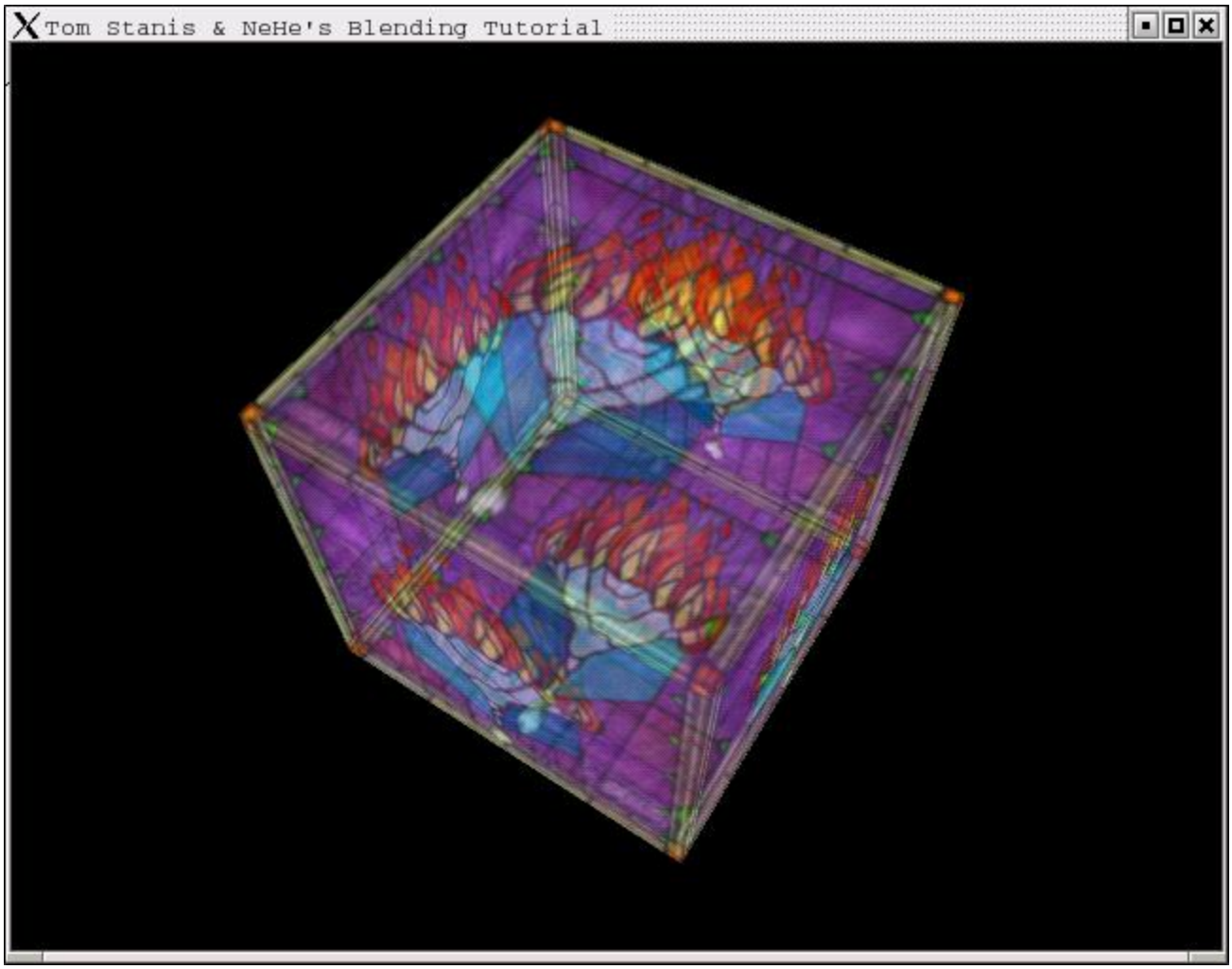
```
  case Qt::Key_F2:
    fullscreen = !fullscreen;
    if ( fullscreen )
    {
```

```
      showFullScreen();
    }
    else
    {
      showNormal();
      setGeometry( 0, 0, 640, 480 );
    }
    update();
    break;
  case Qt::Key_Escape:
    close();
  }
}
```

这一课完了之后，您应该学会创建和使用这三种不同的纹理映射过滤方式。并使用键盘和场景中的对象交互。最后，您应该学会在场景中应用简单的光源，使得场景看起来更逼真。

本课程的源代码。

融合



OpenGL 中的绝大多数特效都与某些类型的（色彩）融合有关。融合的定义为，将某个象素的颜色和已绘制在屏幕上与其对应的象素颜色相互结合。至于如何结合这两个颜色则依赖于颜色的 alpha 通道的分量值，以及/或者所使用的融合函数。alpha 通常是位于颜色值末尾的第 4 个颜色组成分量。前面这些课我们都是用 GL_RGB 来指定颜色的三个分量。相应的 GL_RGBA 可以指定 alpha 分量的值。更进一步，我们可以使用 glColor4f()来代替 glColor3f()。

绝大多数人都认为 alpha 分量代表材料的透明度。这就是说，alpha 值为 0.0 时所代表的材料是完全透明的。alpha 值为 1.0 时所代表的材料则是完全不透明的。

融合的公式

若您对数学不感冒，而只想看看如何实现透明，请跳过这一节。若您想深入理解（色彩）融合的工作原理，这一节应该适合您吧。（CKER 注：其实并不难^-^。原文中的公式如下，CKER 再唠叨一下吧。其实融合的基本原理是就将要分色的图像各象素的颜色以及背景颜色均按照 RGB 规则各自分离之后，根据一图像的 RGB 颜色分量*alpha 值+背景的 RGB 颜色分量*(1-alpha 值)——这样一个简单公式来融合之后，最后将融合得到的 RGB 分量重新合并。）

公式如下：

(Rs Sr + Rd Dr, Gs Sg + Gd Dg, Bs Sb + Bd Db, As Sa + Ad Da)

OpenGL 按照上面的公式计算这两个象素的融合结果。小写的 s 和 r 分别代表源象素和目标象素。大写的 S 和 D 则是相应的融合因子。这些决定了您如何对这些象素融合。绝大多数情况下，各颜色通道的 alpha 融合值大小相同，这样对源象素就有 (As, As, As, As)，目标象素则有 1, 1, 1, 1) - (As, As, As, As)。上面的公式就成了下面的模样:

(Rs As + Rd (1 - As), Gs As + Gd (1 - As), Bs As + Bs (1 - As), As As + Ad (1 - As))

这个公式会生成透明/半透明的效果。

OpenGL 中的融合

在 OpenGL 中实现融合的步骤类似于我们以前提到的 OpenGL 过程。接着设置公式，并在绘制透明对象时关闭写深度缓存。因为我们想在半透明的图形背后绘制对象。这不是正确的混色方法，但绝大多数时候这种做法在简单的项目中都工作的很好。

Rui Martins 的补充：正确的融合过程应该是先绘制全部的场景之后再绘制透明的图形。并且要按照与深度缓存相反的次序来绘制（先画最远的物体）。

考虑对两个多边形（1 和 2）进行 alpha 融合，不同的绘制次序会得到不同的结果。（这里假定多边形 1 离观察者最近，那么正确的过程应该先画多边形 2，再画多边形 1。正如您再现实中所见到的那样，从这两个*透明的*多边形背后照射来的光线总是先穿过多边形 2，再穿过多边形 1，最后才到达观察者的眼睛。）

在深度缓存启用时，您应该将透明图形按照深度进行排序，并在全部场景绘制完毕之后再绘制这些透明物体。否则您将得到不正确的结果。我知道某些时候这样做是很令人痛苦的，但这是正确的方法。

我们要在上一课的代码上进行改动就可以了。

我们将增加一些变量，稍后我们对这些变量进行解释。

NeHeWidget 类

（由 nehewidget.h 展开。）

```
protected:

  bool fullscreen;
```

```
  GLfloat xRot, yRot, zRot;
  GLfloat zoom;
  GLfloat xSpeed, ySpeed;
  GLuint texture[3];
  GLuint filter;

  bool light;
  bool blend;

};
```

比上一课，只增加了 blend 这个变量，说明现在是否使用融合。

（由 nehewidget.cpp 展开。）

```
GLfloat lightAmbient[4] = { 0.5, 0.5, 0.5, 1.0 };
GLfloat lightDiffuse[4] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat lightPosition[4] = { 0.0, 0.0, 2.0, 1.0 };
```

这里定义了三个数组，它们描述的是和光源有关的信息。这里使用的光源和上一课一样。

```
NeHeWidget::NeHeWidget( QWidget* parent, const char* name, bool fs )
    : QGLWidget( parent, name )
{
  xRot = yRot = zRot = 0.0;
  zoom = -5.0;
  xSpeed = ySpeed = 0.0;

  filter = 0;

  light = false;
  blend = false;

  fullscreen = fs;
  setGeometry( 0, 0, 640, 480 );
  setCaption( "Tom Stanis & NeHe's Blending Tutorial" );

  if ( fullscreen )
    showFullScreen();
}
```

我们需要在构造函数中给各个变量赋初值。xRot、yRot、zRot 是 0.0。zoom 是-5.0。xSpeed 和 ySpeed 都是 0。filter 是 0。light 是 false。blend 是 false。

```
void NeHeWidget::loadGLTextures()
{
  QImage tex, buf;
  if ( !buf.load( "./data/Glass.bmp" ) )
  {
    qWarning("Could not read image file, using single-color instead.");
    QImage dummy( 128, 128, 32 );
    dummy.fill( Qt::green.rgb() );
    buf = dummy;
  }
  tex = QGLWidget::convertToGLFormat( buf );

  glGenTextures( 3, &texture[0] );

  glBindTexture( GL_TEXTURE_2D, texture[0] );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
  glTexImage2D( GL_TEXTURE_2D, 0, 3, tex.width(), tex.height(), 0,
      GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() );

  glBindTexture( GL_TEXTURE_2D, texture[1] );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
  glTexImage2D( GL_TEXTURE_2D, 0, 3, tex.width(), tex.height(), 0,
      GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() );

  glBindTexture( GL_TEXTURE_2D, texture[2] );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
GL_LINEAR_MIPMAP_NEAREST );
  gluBuild2DMipmaps( GL_TEXTURE_2D, GL_RGB, tex.width(), tex.height(),
      GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() );
}
```

loadGLTextures()函数就是用来载入纹理的。三个纹理的滤波方式都和上一课一样。

```
void NeHeWidget::paintGL()
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glLoadIdentity();
  glTranslatef(  0.0,  0.0, zoom );

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

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

glBindTexture( GL_TEXTURE_2D, texture[filter] );

glBegin( GL_QUADS );
  glNormal3f( 0.0, 0.0, 1.0 );
  glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );
  glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );
  glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0,  1.0 );
  glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0,  1.0 );

  glNormal3f( 0.0, 0.0, -1.0 );
  glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0, -1.0 );
  glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
  glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );
  glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0, -1.0 );

  glNormal3f( 0.0, 1.0, 0.0 );
  glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
  glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0,  1.0,  1.0 );
  glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0,  1.0,  1.0 );
  glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );

  glNormal3f( 0.0, -1.0, 0.0 );
  glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0, -1.0, -1.0 );
  glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0, -1.0, -1.0 );
  glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );
  glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );

  glNormal3f( 1.0, 0.0, 0.0 );
  glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0, -1.0, -1.0 );
  glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );
  glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0,  1.0,  1.0 );
  glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );

  glNormal3f( -1.0, 0.0, 0.0 );
  glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0, -1.0 );
  glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );
  glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0,  1.0,  1.0 );
  glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
glEnd();

xRot += xSpeed;
yRot += ySpeed;
```

```
}
```

这里也和上一课一样，主要还是画一个立方体。

```
void NeHeWidget::initializeGL()
{
  loadGLTextures();

  glEnable( GL_TEXTURE_2D );
  glShadeModel( GL_SMOOTH );
  glClearColor( 0.0, 0.0, 0.0, 0.5 );
  glClearDepth( 1.0 );
  glEnable( GL_DEPTH_TEST );
  glDepthFunc( GL_LEQUAL );
  glHint( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );

  glLightfv( GL_LIGHT1, GL_AMBIENT, lightAmbient );
  glLightfv( GL_LIGHT1, GL_DIFFUSE, lightDiffuse );
  glLightfv( GL_LIGHT1, GL_POSITION, lightPosition );

  glEnable( GL_LIGHT1 );
```

这里使用光源的方法和前面一样。

```
  glColor4f( 1.0, 1.0, 1.0, 0.5 );
  glBlendFunc( GL_SRC_ALPHA, GL_ONE );
```

上面第一行以全亮度绘制此物体，并对其进行 50%的 alpha 融合（半透明）。当融合选项打开时，此物体将会产生 50%的透明效果。上面第二行设置所采用的融合类型。

Rui Martins 的补充：alpha 通道的值为 0.0 意味着物体材质是完全透明的。1.0 则意味着完全不透明。

```
}
void NeHeWidget::keyPressEvent( QKeyEvent *e )
{
  switch ( e->key() )
  {
  case Qt::Key_L:
    light = !light;
    if ( !light )
    {
      glDisable( GL_LIGHTING );
    }
```

```
    else
    {
      glEnable( GL_LIGHTING );
    }
    updateGL();
    break;
  case Qt::Key_B:
    blend = !blend;
    if ( blend )
    {
      glEnable( GL_BLEND );
      glDisable( GL_DEPTH_TEST );
    }
    else
    {
      glDisable( GL_BLEND );
      glEnable( GL_DEPTH_TEST );
    }
    updateGL();
    break;
```

按下了 B 键，就可以切换是否启用融合方式。

```
  case Qt::Key_F:
    filter += 1;;
    if ( filter > 2 )
    {
      filter = 0;
    }
    updateGL();
    break;
  case Qt::Key_Prior:
    zoom -= 0.2;
    updateGL();
    break;
  case Qt::Key_Next:
    zoom += 0.2;
    updateGL();
    break;
  case Qt::Key_Up:
    xSpeed -= 0.01;
    updateGL();
    break;
  case Qt::Key_Down:
```

```
    xSpeed += 0.01;
    updateGL();
    break;
  case Qt::Key_Right:
    ySpeed += 0.01;
    updateGL();
    break;
  case Qt::Key_Left:
    ySpeed -= 0.01;
    updateGL();
    break;
  case Qt::Key_F2:
    fullscreen = !fullscreen;
    if ( fullscreen )
    {
      showFullScreen();
    }
    else
    {
      showNormal();
      setGeometry( 0, 0, 640, 480 );
    }
    update();
    break;
  case Qt::Key_Escape:
    close();
  }
}
```

但是怎样才能在使用纹理贴图的时候指定融合时的颜色呢？很简单，在调整贴图模式时，文理贴图的每个象素点的颜色都是由 alpha 通道参数与当前地象素颜色相乘所得到的。比如，绘制的颜色是（0.5, 0.6, 0.4），我们会把颜色相乘得到（0.5, 0.6, 0.4, 0.2）（alpha 参数在没有指定时，缺省为零）。

就是如此！OpenGL 实现 alpha 融合的确很简单！

原文注 （1999 年 11 月 13 日）

我（NeHe）融合代码进行了修改，以使显示的物体看起来更逼真。同时对源象素和目的象素使用 alpha 参数来融合，会导致物体的人造痕迹看起来很明显。

会使得物体的背面沿着侧面的地方显得更暗。基本上物体会看起来很怪异。我所用的融合方法也许不是最好的，但的确能够工作。启用光源之后，物体看起来很逼真。感谢 Tom 提供的原始代码，他采用的混色方法是正确的，但物体看起来并不象所期望的那样吸引人:)

代码所作的再次修改是因为在某些显卡上 glDepthMask()函数存在寻址问题。这条命令在某些卡上启用或关闭深度缓冲测试时似乎不是很有效，所以我已经将启用或关闭深度缓冲测试的代码转成老式的 glEnable 和 glDisable。

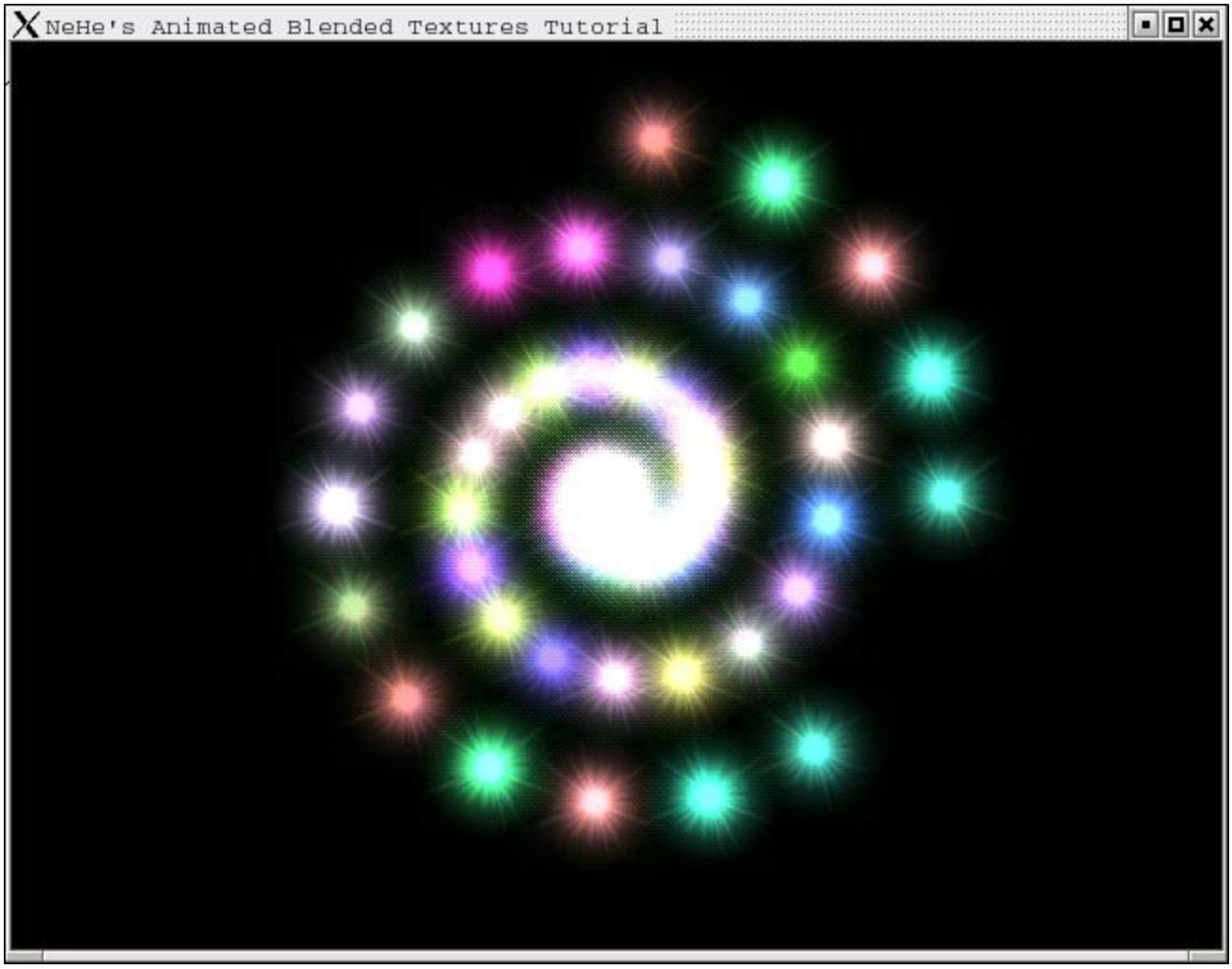
纹理贴图的 alpha 融合

用于纹理贴图的 alpha 参数可以象颜色一样从问题贴图中读取。方法如下，您需要在载入所需的材质同时取得其的 alpha 参数。然后在调用 glTexImage2D()时使用 GL_RGBA 的颜色格式。

本课程的源代码。

[上一课：融合] [Qt OpenGL 教程主页]

在三维空间中移动位图



欢迎进入第九课。到现在为止，您应该很好的理解 OpenGL 了。您已经学会了设置一个 OpenGL 窗口的每个细节。学会在旋转的物体上贴图并打上光线以及融合

（透明）处理。这一课应该算是第一课中级教程。您将学到如下的知识：在三维场景中移动位图，并去除位图上的黑色象素（使用融合）。接着为黑白纹理上色，最后您将学会创建丰富的色彩，并把上过不同色彩的纹理相互融合，得到简单的动画效果。

我们要在上一课的代码上进行改动就可以了。

我们将增加一些变量，稍后我们对这些变量进行解释。

NeHeWidget 类

（由 nehewidget.h 展开。）

```
const GLuint num = 50; //常量 num 存储的是我们绘制的星星的总数


typedef struct        //这个结构是用来存储星星的数据的
{
  int r, g, b;        // r、g、b 存储的是星星的颜色
  GLfloat dist;       // dist 是星星距离中心的距离
  GLfloat angle;      // angle 是星星现在所在的角度
}stars;


protected:

  void initializeGL();
  void paintGL();
  void resizeGL( int width, int height );

  void keyPressEvent( QKeyEvent *e );
  void loadGLTextures();
  void timerEvent( QTimerEvent * );
```

增加了一个 timeEvent(QTimerEvent *)函数，这个函数可以实现整个窗口部件的一些定时操作。

```
protected:

  bool fullscreen;

  GLfloat xRot, yRot, zRot;
  GLfloat zoom;                  // zoom 是星星距离观察者的距离
  GLfloat tilt;                  // tilt 是星星的倾角
```

```
  GLfloat spin;              //spin 是闪烁星星的自转
  GLuint loop;               // loop 是用来绘制所有 50 个星星的全局变量
  GLuint texture[1];         // texture[1]是用来存储纹理的

  bool twinkle;              // twinkle 是用来表示星星是否闪烁

  stars star[num];           // star[num]是用来存储 50 个星星的数据
};
```

（由 nehewidget.cpp 展开。）

```
NeHeWidget::NeHeWidget( QWidget* parent, const char* name, bool fs )
    : QGLWidget( parent, name )
{
  xRot = yRot = zRot = 0.0;
  zoom = -15.0;
  tilt = 90.0;
  spin = 0.0;
  loop = 0;

  twinkle = false;

  fullscreen = fs;
  setGeometry( 0, 0, 640, 480 );
  setCaption( "NeHe's Animated Blended Textures Tutorial" );

  if ( fullscreen )
    showFullScreen();

  startTimer(5);     // startTimer(5)就是每 5 毫秒执行一次 timerEvent()
                        函数做定时操作
}
```

我们需要在构造函数中给各个变量赋初值。

```
void NeHeWidget::loadGLTextures()
{
  QImage tex, buf;
  if ( !buf.load( "./data/Star.bmp" ) )
  {
    qWarning("Could not read image file, using single-color instead.");
    QImage dummy( 128, 128, 32 );
    dummy.fill( Qt::green.rgb() );
```

```
    buf = dummy;
  }
  tex = QGLWidget::convertToGLFormat( buf );

  glGenTextures( 1, &texture[0] );

  glBindTexture( GL_TEXTURE_2D, texture[0] );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
  glTexImage2D( GL_TEXTURE_2D, 0, 3, tex.width(), tex.height(), 0,
      GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() );
}
```

loadGLTextures()函数就是用来载入纹理的。

```
void NeHeWidget::initializeGL()
{
  loadGLTextures();
  glEnable( GL_TEXTURE_2D );
  glShadeModel( GL_SMOOTH );
  glClearColor( 0.0, 0.0, 0.0, 0.5 );
  glClearDepth( 1.0 );
  glEnable( GL_DEPTH_TEST );
  glDepthFunc( GL_LEQUAL );
  glHint( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
  glBlendFunc( GL_SRC_ALPHA, GL_ONE );
  glEnable( GL_BLEND );
```

上述设置前面的各章都提到过了，就不再讲了。

```
  for ( loop = 0; loop < num; loop++ )
  {
    star[loop].angle = 0.0;
    star[loop].dist = ( float(loop)/num ) * 5.0;
    star[loop].r = rand() % 256;
    star[loop].g = rand() % 256;
    star[loop].b = rand() % 256;
  }
```

设置了每颗星星的起始角度、距离和颜色。您会注意到修改结构的属性有多容易。全部 50 颗星星都会被循环设置。要改变 star[1]的角度我们所要做的只是 star[1].angle={某个数值}；就这么简单！

第 loop 颗星星离中心的距离是将 loop 的值除以星星的总颗数，然后乘上 5.0。基本上这样使得后一颗星星比前一颗星星离中心更远一点。这样当 loop 为 50 时(最

后一颗星星），loop 除以 num 正好是 1.0。之所以要乘以 5.0 是因为 1.0*5.0 就是 5.0。5.0 已经很接近屏幕边缘。我不想星星飞出屏幕，5.0 是最好的选择了。当然如果如果您将场景设置的更深入屏幕里面的话，也许可以使用大于 5.0 的数值，但星星看起来就更小一些（都是透视的缘故）。

您还会注意到每颗星星的颜色都是从 0～255 之间的一个随机数。也许您会奇怪为何这里的颜色得取值范围不是 OpenGL 通常的 0.0～1.0 之间。这里我们使用的颜色设置函数是 glColor4ub，而不是以前的 glColor4f。ub 意味着参数是 unsigned byte 型的。一个 byte 的取值范围是 0～255。这里使用 byte 值取随机整数似乎要比取一个浮点的随机数更容易一些。

```
}
void NeHeWidget::paintGL()
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
```

清楚屏幕及深度缓存。

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

选择纹理。

```
  for ( loop = 0; loop < num; loop++ )
  {
```

这段程序我们来循环绘制所有的星星。

```
    glLoadIdentity();
```

绘制每颗星星之前，重置模型观察矩阵。

```
    glTranslatef( 0.0, 0.0, zoom );
```

深入屏幕里面（使用“zoom”的值）。

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

倾斜视角（使用“tilt”的值）。

```
    glRotatef( star[loop].angle, 0.0, 1.0, 0.0 );
```

接下来的代码，我们来移动星星。星星开始时位于屏幕的中心。我们要做的第一件事是把场景沿 Y 轴旋转。如果我们旋转 90 度的话，X 轴不再是自左至右的了，他将由里向外穿出屏幕。为了让大家更清楚些，举个例子。假想您站在房子中间。再设想您左侧的墙上写着-x，前面的墙上写着-z，右面墙上就是+x 咯，您身后的墙上则是+z。加入整个房子向右转 90 度，但您没有动，那么前面的墙上将是-x

而不再是-z 了。所有其他的墙也都跟着移动。-z 出现在右侧，+z 出现在左侧，+x 出现在您背后。神经错乱了吧？通过旋转场景，我们改变了 x 和 z 平面的方向。

```
glTranslatef( star[loop].dist, 0.0, 0.0 );
```

这代码沿 x 轴移动一个正值。通常 x 轴上的正值代表移向了屏幕的右侧（也就是通常的 x 轴的正向），但这里由于我们绕 y 轴旋转了坐标系，x 轴的正向可以是任意方向。如果我们转 180 度的话，屏幕的左右侧就镜像反向了。因此，当我们沿 x 轴正向移动时，可能向左，向右，向前或向后。

```
glRotatef( -star[loop].angle, 0.0, 1.0, 0.0 );
glRotatef( -tilt, 1.0, 0.0, 0.0 );
```

星星实际上是一个平面的纹理。现在您在屏幕中心画了个平面的四边形然后贴上纹理，这看起来很不错。一切都如您所想的那样。但是当您当您沿着 y 轴转上个 90 度的话，纹理在屏幕上就只剩右侧和左侧的两条边朝着您。看起来就是一条细线。这不是我们所想要的。我们希望星星永远正面朝着我们，而不管屏幕如何旋转或倾斜。

我们通过在绘制星星之前，抵消对星星所作的任何旋转来实现这个愿望。您可以采用逆序来抵消旋转。当我们倾斜屏幕时，我们实际上以当前角度旋转了星星。通过逆序，我们又以当前角度“反旋转”星星。也就是以当前角度的负值来旋转星星。就是说，如果我们将星星旋转了 10 度的话，又将其旋转-10 度来使星星在那个轴上重新面对屏幕。上面的第一行抵消了沿 y 轴的旋转。然后，我们还需要抵消掉沿 x 轴的屏幕倾斜。要做到这一点，我们只需要将屏幕再旋转-tilt 倾角。在抵消掉 x 和 y 轴的旋转后，星星又完全面对着我们了。

```
if ( twinkle )
{
  glColor4ub( star[(num-loop)-1].r,
     star[(num-loop)-1].g,
     star[(num-loop)-1].b, 255 );
  glBegin( GL_QUADS );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0, 0.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( 1.0, -1.0, 0.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( 1.0, 1.0, 0.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0, 1.0, 0.0 );
  glEnd();
}
```

如果 twinkle 为真，我们在屏幕上先画一次不旋转的星星：将星星总数（num）减去当前的星星数（loop）再减去 1，来提取每颗星星的不同颜色（这么做是因为循环范围从 0 到 num-1）。举例来说，结果为 10 的时候，我们就使用 10 号星

星的颜色。这样相邻星星的颜色总是不同的。这不是个好法子，但很有效。最后一个值是 alpha 通道分量。这个值越小，这颗星星就越暗。

由于启用了 twinkle，每颗星星最后会被绘制两遍。程序运行起来会慢一些，这要看您的机器性能如何了。但两遍绘制的星星颜色相互融合，会产生很棒的效果。同时由于第一遍的星星没有旋转，启用 twinkle 后的星星看起来有一种动画效果。（如果您这里看不懂得话，就自己去看程序的运行效果吧。）

值得注意的是给纹理上色是件很容易的事。尽管纹理本身是黑白的，纹理将变成我们在绘制它之前选定的任意颜色。此外，同样值得注意的是我们在这里使用的颜色值是 byte 型的，而不是通常的浮点数。甚至 alpha 通道分量也是如此。

```
    glRotatef( spin, 0.0, 0.0, 1.0 );
    glColor4ub( star[loop].r, star[loop].g, star[loop].b, 255 );
    glBegin( GL_QUADS );
      glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0, 0.0 );
      glTexCoord2f( 1.0, 0.0 ); glVertex3f( 1.0, -1.0, 0.0 );
      glTexCoord2f( 1.0, 1.0 ); glVertex3f( 1.0, 1.0, 0.0 );
      glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0, 1.0, 0.0 );
    glEnd();
```

上面是绘制第二遍的星星。唯一和前面的代码不同的是这一遍的星星肯定会被绘制，并且这次的星星绕着 z 轴旋转。

```
    spin += 0.01;
    star[loop].angle += float(loop)/num;
    star[loop].dist -= 0.01;
```

以上的代码代表星星的运动。我们增加 spin 的值来旋转所有的星星（公转）。然后，将每颗星星的自转角度增加 loop/num。这使离中心更远的星星转的更快。最后减少每颗星星离屏幕中心的距离。这样看起来，星星们好像被不断地吸入屏幕的中心。

```
    if ( star[loop].dist < 0.0 )
    {
      star[loop].dist += 5.0;
      star[loop].r = rand() % 256;
      star[loop].g = rand() % 256;
      star[loop].b = rand() % 256;
    }
  }
```

检查星星是否已经碰到了屏幕中心。当星星碰到屏幕中心时，我们为它赋一个新颜色，然后往外移 5 个单位，这颗星星将踏上它回归屏幕中心的旅程。

```
}
void NeHeWidget::timerEvent(QTimerEvent*)
{
  updateGL();
}
```

这里就是定时操作函数 timerEvent()，执行的操作就是 updateGL()，就是刷新窗口了，其实它也会调用 paintGL()，所以就实现了每 5 毫秒刷新一次的动画效果。

```
void NeHeWidget::keyPressEvent( QKeyEvent *e )
{
  switch ( e->key() )
  {
  case Qt::Key_T:
    twinkle = !twinkle;
    updateGL();
    break;
```

按下了 T 键，就可以切换是否闪烁。

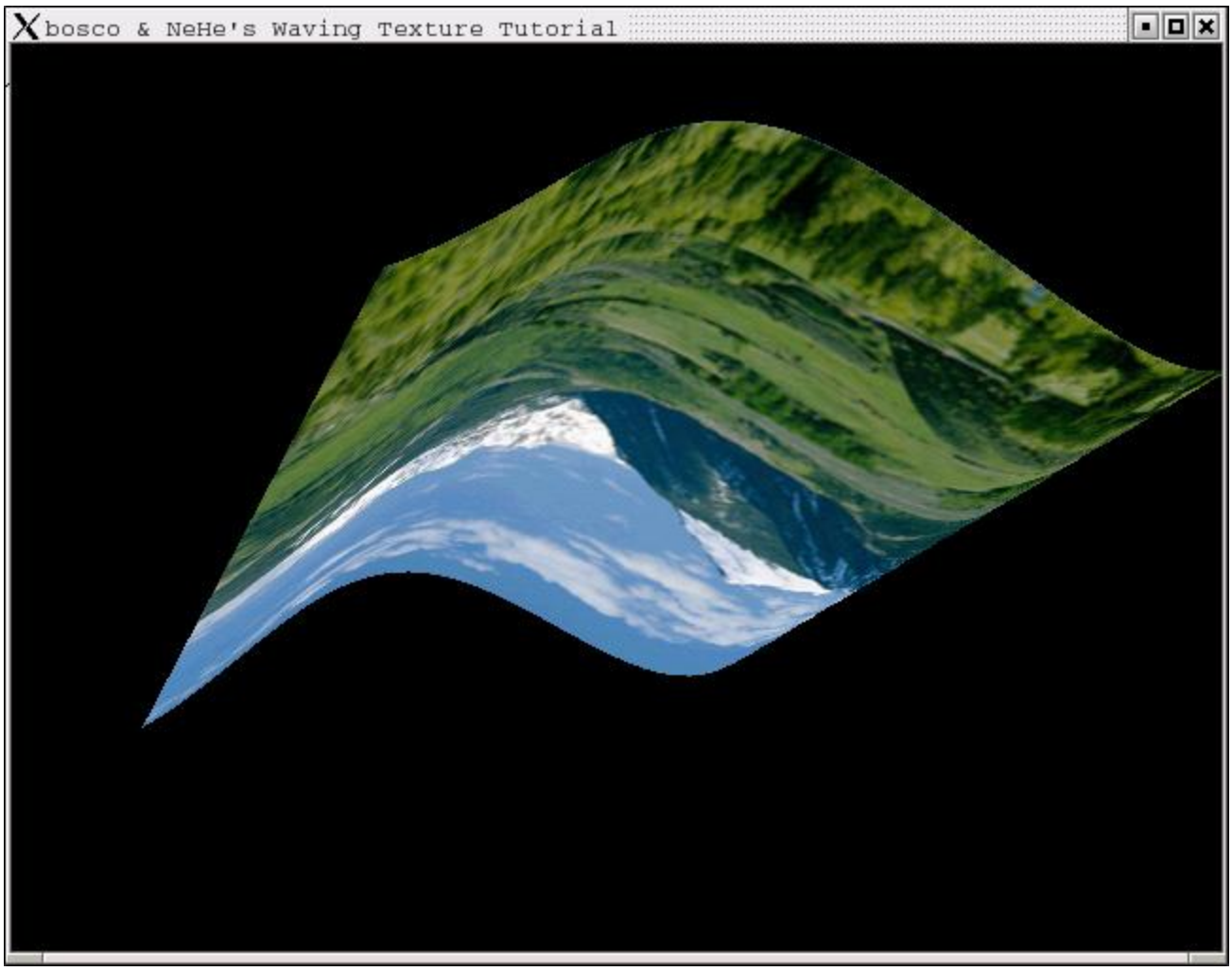
```
  case Qt::Key_Up:
    tilt -= 0.5;
    updateGL();
    break;
  case Qt::Key_Down:
    tilt += 0.5;
    updateGL();
    break;
  case Qt::Key_Prior:
    zoom -= 0.2;
    updateGL();
    break;
  case Qt::Key_Next:
    zoom += 0.2;
    updateGL();
    break;
  case Qt::Key_F2:
    fullscreen = !fullscreen;
    if ( fullscreen )
    {
      showFullScreen();
    }
    else
    {
      showNormal();
```

```
      setGeometry( 0, 0, 640, 480 );
    }
    update();
    break;
  case Qt::Key_Escape:
    close();
  }
}
```

这一课我尽我所能来解释如何加载一个灰阶位图纹理，（使用融合）去掉它的背景色后，再给它上色，最后让它在三维场景中移动。我已经向您展示了如何创建漂亮的颜色与动画效果。实现原理是在原始位图上再重叠一份位图拷贝。到现在为止，只要您很好的理解了我所教您的一切，您应该已经能够毫无问题的制作您自己的 3D Demo 了。所有的基础知识都已包括在内！

本课程的源代码。

旗的效果（波动纹理）



大家好！对那些想知道我在这里作了些什么的朋友，您可以先按文章的末尾所列出的链接，下载我那毫无意义的 Demo 看看先！我是 bosco，我将尽我所能教您来实现一个以正弦波方式运动的图象。这一课基于 NeHe 的教程第六课，当然您至少也应该学会了一至六课的知识。您需要下载源码压缩包，并将压缩包内带的 data 目录连其下的位图一起释放至您的代码目录下。或者使用您自己的位图，当然它的尺寸必须适合 OpenGL 纹理的要求。

NeHeWidget 类

（由 nehewidget.h 展开。）

```
class NeHeWidget : public QGLWidget
{
    Q_OBJECT

public:

  NeHeWidget( QWidget* parent = 0, const char* name = 0, bool fs = false );
  ~NeHeWidget();

protected:

  void initializeGL();
  void paintGL();
  void resizeGL( int width, int height );

  void keyPressEvent( QKeyEvent *e );
  void loadGLTextures();
  void timerEvent( QTimerEvent * );
```

这些函数前几课中都提到过了。

```
protected:

  bool fullscreen;

  GLfloat xRot, yRot, zRot;
  GLfloat hold;
  GLuint texture[1];

  float points[45][45][3];
  int wiggle_count;
};
```

我们将使用 points 数组来存放网格各顶点独立的(x,y,z)坐标。这里网格由 45×45 点形成，换句话说也就是由 44 格×44 格的小方格子依次组成了。wiggle_count 用来指定纹理波浪的运动速度，每 3 帧一次看起来很不错。变量 hold 将存放一个用来对旗形波浪进行光滑的浮点数。

（由 nehewidget.cpp 展开。）

```
#include <math.h>
```

因为我们在程序中要使用到 sin 和 cos 两个三角函数，所以我们需要包含 math.h 这个头文件。

```
NeHeWidget::NeHeWidget( QWidget* parent, const char* name, bool fs )
    : QGLWidget( parent, name )
{
  xRot = yRot = zRot = 0.0;
  hold = 0.0;

  wiggle_count = 0;

  fullscreen = fs;
  setGeometry( 0, 0, 640, 480 );
  setCaption( "bosco & NeHe's Waving Texture Tutorial" );

  if ( fullscreen )
    showFullScreen();

  startTimer( 5 );
}
```

我们需要在构造函数中给各个变量赋初值。startTimer()函数我们在第九课中已经讲过了。

```
void NeHeWidget::loadGLTextures()
{
  QImage tex, buf;
  if ( !buf.load( "./data/Tim.bmp" ) )
  {
    qWarning( "Could not read image file, using single-color instead." );
    QImage dummy( 128, 128, 32 );
    dummy.fill( Qt::green.rgb() );
    buf = dummy;
  }
  tex = QGLWidget::convertToGLFormat( buf );
```

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

  glBindTexture( GL_TEXTURE_2D, texture[0] );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  glTexImage2D( GL_TEXTURE_2D, 0, 3, tex.width(), tex.height(), 0,
      GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() );
}
```

loadGLTextures()函数就是用来载入纹理的。

```
void NeHeWidget::initializeGL()
{
  loadGLTextures();

  glEnable( GL_TEXTURE_2D );
  glShadeModel( GL_SMOOTH );
  glClearColor( 0.0, 0.0, 0.0, 0.5 );
  glClearDepth( 1.0 );
  glEnable( GL_DEPTH_TEST );
  glDepthFunc( GL_LEQUAL );
  glHint( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
  glPolygonMode( GL_BACK, GL_FILL );
  glPolygonMode( GL_FRONT, GL_LINE );
```

上面的代码指定使用完全填充模式来填充多边形区域的背面（或者叫做后表面吧）。相反，多边形的正面（前表面）则使用轮廓线填充了。这些方式完全取决于您的个人喜好。并且与多边形的方位或者顶点的方向有关。详情请参考 Red Book。这里我顺便推销一本推动我学习 OpenGL 的好书－Addison-Wesley 出版的《Programmer's Guide to OpenGL》。个人以为这是学习 OpenGL 的无价之宝。

```
  for ( int x = 0; x < 45; x++ )
  {
    for ( int y = 0; y < 45; y++ )
    {
      points[x][y][0] = float( ( x/5.0 ) - 4.5 );
      points[x][y][1] = float( ( y/5.0 ) - 4.5 );
      points[x][y][2] = float( sin( ( ( ( x/5.0 ) * 40.0 )/360.0 ) *
                                          3.141592654 * 2.0 ) );
    }
  }
```

这里感谢 Graham Gibbons 关于使用整数循环变量消除波浪间的脉冲锯齿的建议。

上面的两个循环初始化网格上的点。使用整数循环可以消除由于浮点运算取整造成的脉冲锯齿的出现。我们将 x 和 y 变量都除以 5，再减去 4.5。这样使得我们的波浪可以“居中”（这样计算所得结果将落在区间[-4.5，4.5]之间）。

点[x][y][2]最后的值就是一个 sin 函数计算的结果。sin()函数需要一个弧度参变量。将 float_x 乘以 40.0，得到角度值。然后除以 360.0 再乘以 PI，乘以 2.0，就转换为弧度了。

```
}
void NeHeWidget::paintGL()
{
  int x, y;
  float float_x, float_y, float_xb, float_yb;
```

x、y 是循环变量，float_x、float_y、float_xb、float_yb 是用来将旗形的波浪分割成很小的四边形。

```
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glLoadIdentity();

  glTranslatef( 0.0, 0.0, -12.0 );

  glRotatef( xRot, 1.0, 0.0, 0.0 );
  glRotatef( yRot, 0.0, 1.0, 0.0 );
  glRotatef( zRot, 0.0, 0.0, 1.0 );

  glBindTexture( GL_TEXTURE_2D, texture[0] );

  glBegin( GL_QUADS );
```

开始绘制四边形。

```
    for ( x = 0; x < 44; x++ )
    {
```

沿 X 平面 0-44 循环（45 点）

```
      for ( y = 0; y < 44; y++ )
      {
```

沿 Y 平面 0-44 循环（45 点）

```
        float_x = float(x)/44.0;
        float_y = float(y)/44.0;
        float_xb = float(x+1)/44.0;
```

```
        float_yb = float(y+1)/44.0;
```

上面我们使用 4 个变量来存放纹理坐标。每个多边形（网格之间的四边形）分别映射了纹理的 1/44 x 1/44 部分。循环首先确定左下顶点的值，然后我们据此得到其他三点的值。

```
        glTexCoord2f( float_x, float_y );
        glVertex3f( points[x][y][0], points[x][y][1],
points[x][y][2] );

        glTexCoord2f( float_x, float_yb );
        glVertex3f( points[x][y+1][0], points[x][y+1][1],
points[x][y+1][2] );

        glTexCoord2f( float_xb, float_yb );
        glVertex3f( points[x+1][y+1][0], points[x+1][y+1][1],
points[x+1][y+1][2] );

        glTexCoord2f( float_xb, float_y );
        glVertex3f( points[x+1][y][0], points[x+1][y][1],
points[x+1][y][2] );
```

上面四个坐标分别为左下、左上、右上、右下。

上面几行使用 glTexCoord2f()和 glVertex3f()载入数据。提醒一点：四边形是逆时针绘制的。这就是说，您开始所见到的表面是背面。后表面完全填充了，前表面由线条组成。

如果您按顺时针顺序绘制的话，您初始时见到的可能是前表面。也就是说您将看到网格型的纹理效果而不是完全填充的。

```
      }
    }
  glEnd();
```

四边形绘制结束。

```
  if ( wiggle_count == 2 )
  {
    for ( y = 0; y < 45; y++ )
    {
      hold = points[0][y][2];
      for ( x = 0; x < 44; x++ )
      {
        points[x][y][2] = points[x+1][y][2];
```

```
      }
      points[44][y][2] = hold;
    }
    wiggle_count = 0;
  }

  wiggle_count++;
```

上面所作的事情是先存储每一行的第一个值，然后将波浪左移一下，是图象产生波浪。存储的数值挪到末端以产生一个永无尽头的波浪纹理效果。然后重置计数器 wiggle_count 以保持动画的进行。

上面的代码由 NeHe（2000 年 2 月）修改过，以消除波浪间出现的细小锯齿。

```
  xRot += 0.3;
  yRot += 0.2;
  zRot += 0.4;
```

标准的 NeHe 旋转增量:)。

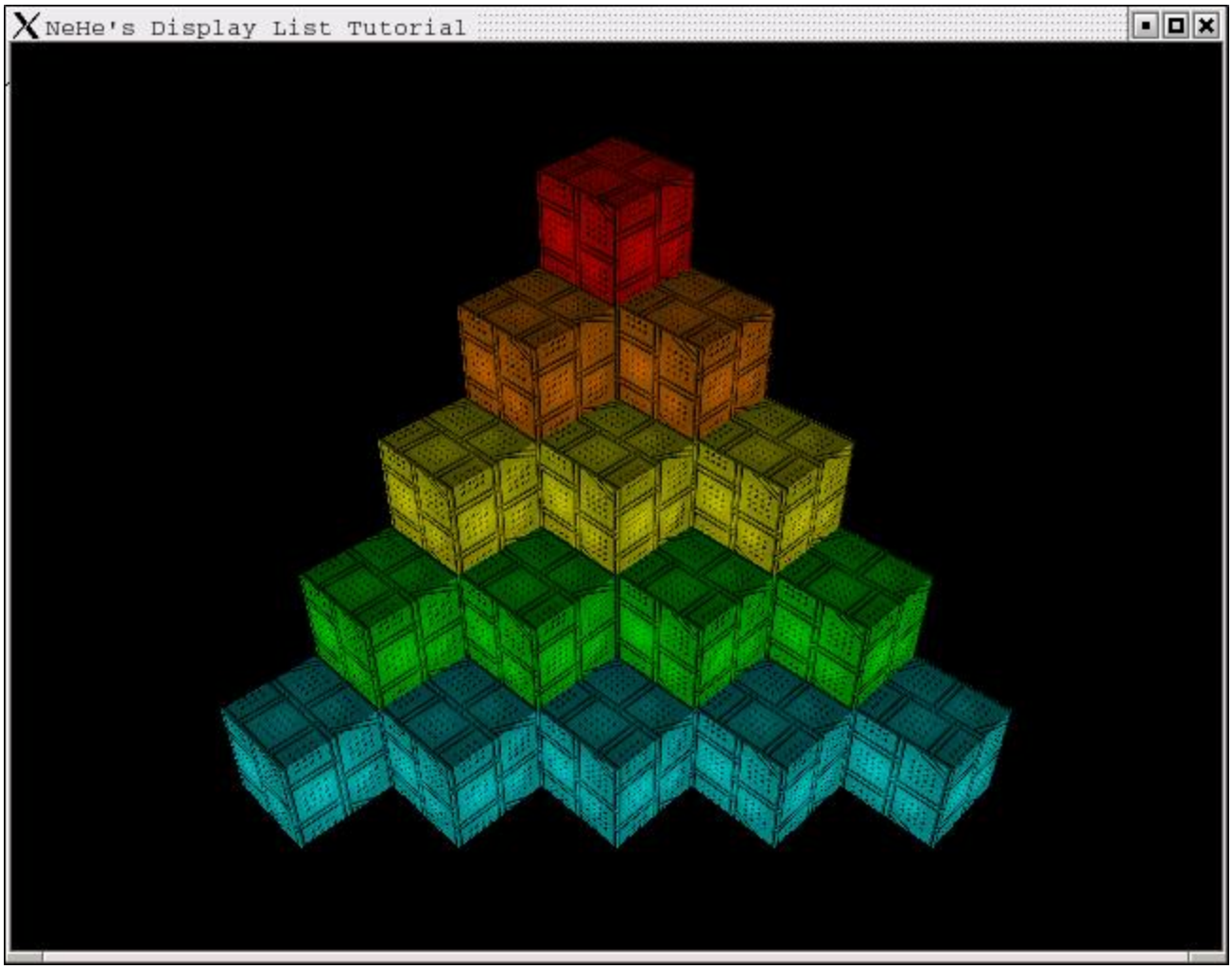
```
}
void NeHeWidget::timerEvent(QTimerEvent*)
{
  updateGL();
}


void NeHeWidget::keyPressEvent( QKeyEvent *e )
{
  switch ( e->key() )
  {
  case Qt::Key_F2:
    fullscreen = !fullscreen;
    if ( fullscreen )
    {
      showFullScreen();
    }
    else
    {
      showNormal();
      setGeometry( 0, 0, 640, 480 );
    }
    update();
    break;
```

```
  case Qt::Key_Escape:
    close();
  }
}
```

现在编译并运行程序，您将看到一个漂亮的位图波浪。除了嘘声一片之外，我不敢确信大家的反应。但我希望大家能从这一课中学到点什么。如果您有任何问题或者需要澄清的地方，请随便联络我。感谢大家！

本课程的源代码。

显示列表



这次我将教你如何使用显示列表，显示列表将加快程序的速度，而且可以减少代码的长度。

当你在制作游戏里的小行星场景时，每一层上至少需要两个行星，你可以用 OpenGL 中的多边形来构造每一个行星。聪明点的做法是做一个循环，每个循环

画出行星的一个面，最终你用几十条语句画出了一个行星。每次把行星画到屏幕上都是很困难的。当你面临更复杂的物体时你就会明白了。

那么，解决的办法是什么呢？用显示列表，你只需要一次性建立物体，你可以贴图，用颜色，想怎么弄就怎么弄。给显示列表一个名字，比如给小行星的显示列表命名为“asteroid”。现在，任何时候我想在屏幕上画出行星，我只需要调用 glCallList(asteroid)。之前做好的小行星就会立刻显示在屏幕上了。因为小行星已经在显示列表里建造好了，OpenGL 不会再计算如何构造它。它已经在内存中建造好了。这将大大降低 CPU 的使用，让你的程序跑的更快。

那么，开始学习咯。我称这个 DEMO 为 Q-Bert 显示列表。最终这个 DEMO 将在屏幕上画出 15 个立方体。每个立方体都由一个盒子和一个顶构成，顶部是一个单独的显示列表，盒子没有顶。

这一课是建立在第六课的基础上的，我将重写大部分的代码，这样容易看懂。下面的这些代码在所有的课程中差不多都用到了。

NeHeWidget 类

（由 nehewidget.h 展开。）

```
class NeHeWidget : public QGLWidget
{
    Q_OBJECT

public:

  NeHeWidget( QWidget* parent = 0, const char* name = 0, bool fs = false );
  ~NeHeWidget();

protected:

  void initializeGL();
  void paintGL();
  void resizeGL( int width, int height );

  void keyPressEvent( QKeyEvent *e );
  void loadGLTextures();
  void buildLists();
```

这个是建立显示列表的函数。

```
protected:
```

```
  bool fullscreen;

  GLfloat xRot, yRot, zRot;
  GLuint box, top;
```

这里是两个用来存放显示列表的指针。

```
  GLuint xLoop, yLoop;
```

这里是两个表示立方体位置的变量。

```
  GLuint texture[1];
};
```

（由 nehewidget.cpp 展开。）

```
static GLfloat boxcol[5][3] =
{
  { 1.0, 0.0, 0.0 },
  { 1.0, 0.5, 0.0 },
  { 1.0, 1.0, 0.0 },
  { 0.0, 1.0, 0.0 },
  { 0.0, 1.0, 1.0 }
};

static GLfloat topcol[5][3] =
{
  { 0.5, 0.0, 0.0 },
  { 0.5, 0.25, 0.0 },
  { 0.5, 0.5, 0.0 },
  { 0.0, 0.5, 0.0 },
  { 0.0, 0.5, 0.5 }
};
```

这里是两个颜色数组。

```
NeHeWidget::NeHeWidget( QWidget* parent, const char* name, bool fs )
    : QGLWidget( parent, name )
{
  xRot = yRot = zRot = 0.0;
  box = top = 0;

  xLoop = yLoop = 0;

  fullscreen = fs;
```

```
  setGeometry( 0, 0, 640, 480 );
  setCaption( "NeHe's Display List Tutorial" );

  if ( fullscreen )
    showFullScreen();
}
```

我们需要在构造函数中给各个变量赋初值。

```
void NeHeWidget::loadGLTextures()
{
  QImage tex, buf;
  if ( !buf.load( "./data/Cube.bmp" ) )
  {
    qWarning( "Could not read image file, using single-color instead." );
    QImage dummy( 128, 128, 32 );
    dummy.fill( Qt::green.rgb() );
    buf = dummy;
  }
  tex = QGLWidget::convertToGLFormat( buf );

  glGenTextures( 1, &texture[0] );

  glBindTexture( GL_TEXTURE_2D, texture[0] );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  glTexImage2D( GL_TEXTURE_2D, 0, 3, tex.width(), tex.height(), 0,
      GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() );
}
```

loadGLTextures()函数就是用来载入纹理的。

贴图纹理的代码和之前教程里的代码是一样的。我们需要一个可以贴在立方体上的纹理。我决定使用 mipmapping 处理让纹理看上去光滑，因为我讨厌看见像素点。纹理的文件名是“Cube.bmp”，存放在 data 目录下。

```
void NeHeWidget::buildLists()
{
  box = glGenLists( 2 );
```

开始的时候我们告诉 OpenGL 我们要建立两个显示列表。glGenLists(2)建立了两个显示列表的空间，并返回第一个显示列表的指针。“box”指向第一个显示列表，任何时候调用“box”第一个显示列表就会显示出来。

现在开始构造第一个显示列表。我们已经申请了两个显示列表的空间了，并且有box 指针指向第一个显示列表。所以现在我们应该告诉 OpenGL 要建立什么类型的显示列表。

```
glNewList( box, GL_COMPILE );
```

我们用 glNewList()命令来做这个事情。你一定注意到了 box 是第一个参数，这表示 OpenGL 将把列表存储到 box 所指向的内存空间。第二个参数 GL_COMPILE 告诉 OpenGL 我们想预先在内存中构造这个列表，这样每次画的时候就不必重新计算怎么构造物体了。

GL_COMPILE 类似于编程。在你写程序的时候，把它装载到编译器里，你每次运行程序都需要重新编译。而如果它已经编译成了.exe 文件，那么每次你只需要点击那个.exe 文件就可以运行它了，不需要编译。当 OpenGL 编译过显示列表后，就不需要再每次显示的时候重新编译它了。这就是为什么用显示列表可以加快速度。

你可以在 glNewList()和 glEngList()中间加上任何你想加上的代码。可以设置颜色，贴图等等。唯一不能加进去的代码就是会改变显示列表的代码。显示列表一旦建立，你就不能改变它。

比如你想加上 glColor3ub(rand()%255, rand()%255, rand()%255)，使得每一次画物体时都会有不同的颜色。但因为显示列表只会建立一次，所以每次画物体的时候颜色都不会改变。物体将会保持第一次建立显示列表时的颜色。 如果你想改变显示列表的颜色，你只有在调用显示列表之前改变颜色。后面将详细解释这一点。

```
glBegin( GL_QUADS );
```

这部分的代码画出一个没有顶部的盒子，它不会出现在屏幕上，只会存储在显示列表里。

```
    glNormal3f( 0.0, -1.0, 0.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0, -1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );

    glNormal3f( 0.0, 0.0, 1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0,  1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0,  1.0 );

    glNormal3f( 0.0, 0.0, -1.0 );
```

```
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0, -1.0 );

    glNormal3f( 1.0, 0.0, 0.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0,  1.0,  1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );

    glNormal3f( -1.0, 0.0, 0.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0,  1.0,  1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
  glEnd();

  glEndList();
```

用 glEngList()命令，我们告诉 OpenGL 我们已经完成了一个显示列表。在 glNewList()和 glEngList()之间的任何东西就是显示列表的一部分。

```
  top = box + 1;
```

现在我们来建立第二个显示列表。在上一个显示列表的指针上加 1，就得到了第二个显示列表的指针。第二个显示列表的指针命名为“top”。

```
  glNewList( top, GL_COMPILE );

  glBegin( GL_QUADS );
```

这部分代码画出盒子的顶部。

```
    glNormal3f( 0.0, 1.0, 0.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0,  1.0,  1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0,  1.0,  1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );
  glEnd();

  glEndList();
```

然后告诉 OpenGL 第二个显示列表建立完毕。

```
}
void NeHeWidget::initializeGL()
{
  loadGLTextures();
  buildLists();
```

请注意代码的顺序，先读入纹理，然后建立显示列表，这样当我们建立显示列表的时候就可以将纹理贴到立方体上了。

```
  glEnable( GL_TEXTURE_2D );
  glShadeModel( GL_SMOOTH );
  glClearColor( 0.0, 0.0, 0.0, 0.5 );
  glClearDepth( 1.0 );
  glEnable( GL_DEPTH_TEST );
  glDepthFunc( GL_LEQUAL );
  glEnable( GL_LIGHT0 );
  glEnable( GL_LIGHTING );
  glEnable( GL_COLOR_MATERIAL );
```

上面的三行使灯光有效。Light0 一般来说是在显卡中预先定义过的，如果 Light0 不工作，把下面那行注释掉好了。

最后一行的 GL_COLOR_MATERIAL 使我们可以用颜色来贴纹理。如果没有这行代码，纹理将始终保持原来的颜色，glColor3f(r, g, b)就没有用了。总之这行代码是很有用的。

```
  glHint( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );
```

最后，设置投影校正。

```
}
```

现在来看绘画的代码。对数学我从来都是很头大的，没有 sin，没有 cos，但仍然看起来很奇怪（相信读者不会觉得头大）。首先，按惯例，清除屏幕和深度缓冲。

然后捆绑纹理到立方体上（我知道捆绑这个词不太专业，但是……）。可以将这行放在显示列表里，但放在外边，就可以在任何时候修改它。

```
void NeHeWidget::paintGL()
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glBindTexture( GL_TEXTURE_2D, texture[0] );
```

现在到了真正有趣的地方了。用一个循环，循环变量用于改变 Y 轴位置，在 Y 轴上画 5 个立方体，所以用从 1 到 5 的循环。

```
  for ( yLoop = 1; yLoop < 6; yLoop++ )
  {
```

另外用一个循环，循环变量用于改变 X 轴位置。每行上的立方体数目取决于行数，所以循环方式如下。

```
    for ( xLoop = 0; xLoop < yLoop; xLoop++ )
    {
```

下边的代码是移动和旋转当前坐标系到需要画出立方体的位置。（原文有很罗嗦的一大段，相信大家的数学功底都不错，就不翻译了）

```
      glLoadIdentity();
      glTranslatef( 1.4 + (float(xLoop) * 2.8) - (float(yLoop) * 1.4),
          ( (6.0 - (float(yLoop)) ) * 2.4 ) - 7.0, -20.0 );
      glRotatef( 45.0 - (2.0 * yLoop) + xRot, 1.0, 0.0, 0.0 );
      glRotatef( 45.0 + yRot, 0.0, 1.0, 0.0 );
```

然后在正式画盒子之前设置颜色。每个盒子用不同的颜色。

```
      glColor3fv( boxcol[yLoop-1] );
```

好了，颜色设置好了。现在需要做的就是画出盒子。不用写出画多边形的代码，只需要用 glCallList(box)命令调用显示列表。盒子将会用 glColor3fv()所设置的颜色画出来。

```
      glCallList( box );
```

然后用另外的颜色画顶部。搞定。

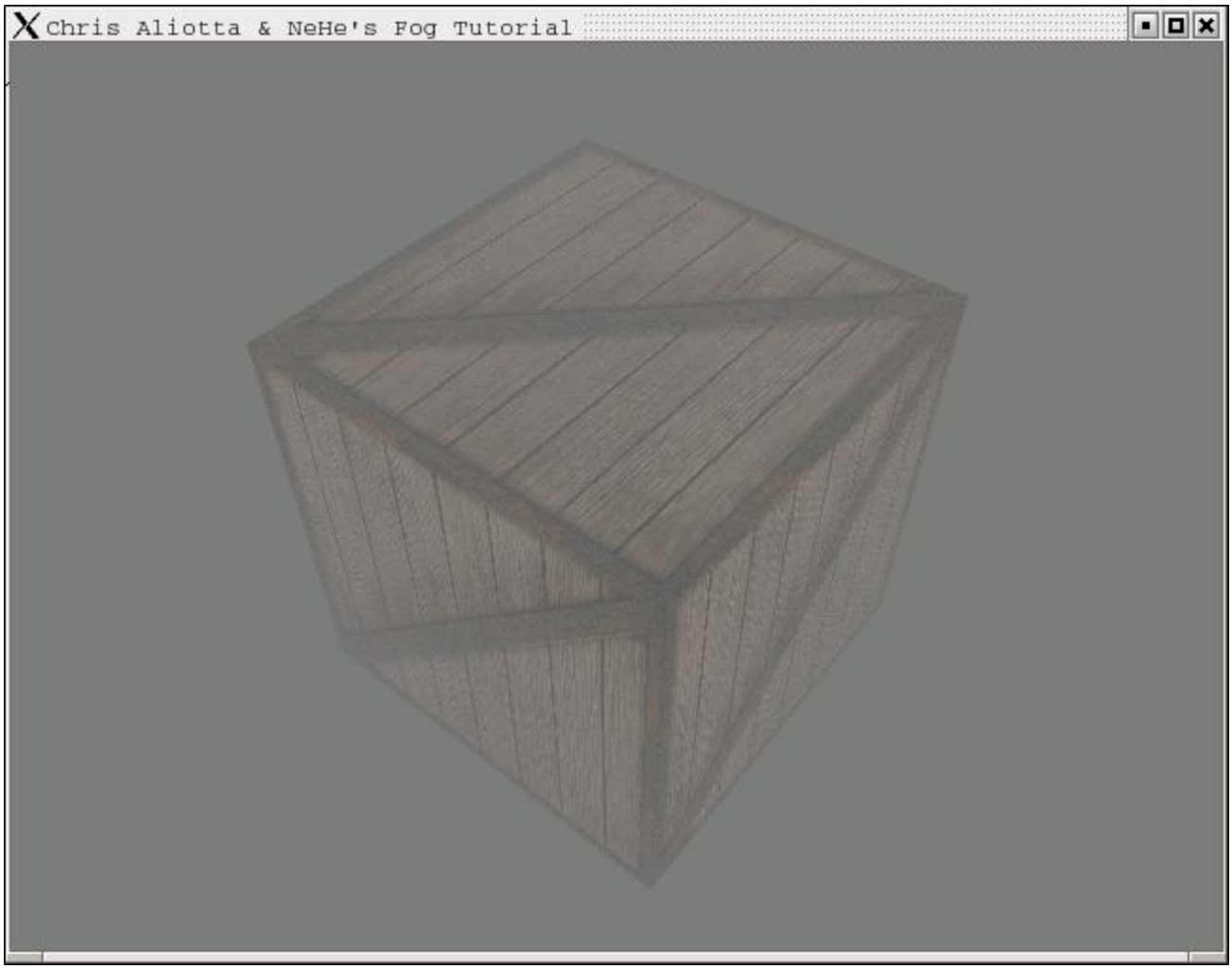
```
      glColor3fv( topcol[yLoop-1] );
      glCallList( top );
    }
  }
}
void NeHeWidget::keyPressEvent( QKeyEvent *e )
{
  switch ( e->key() )
  {
  case Qt::Key_Up:
    xRot -= 0.2;
    updateGL();
    break;
  case Qt::Key_Down:
    xRot += 0.2;
```

```
    updateGL();
    break;
  case Qt::Key_Left:
    yRot -= 0.2;
    updateGL();
    break;
  case Qt::Key_Right:
    yRot += 0.2;
    updateGL();
    break;
  case Qt::Key_F2:
    fullscreen = !fullscreen;
    if ( fullscreen )
    {
      showFullScreen();
    }
    else
    {
      showNormal();
      setGeometry( 0, 0, 640, 480 );
    }
    update();
    break;
  case Qt::Key_Escape:
    close();
  }
}
```

上面就是键盘控制，用上下左右键来控制立方体的运动。

本课程的源代码。

看起来很棒的雾



难道你不想把“雾”加入到你的 OpenGL 程序中吗？那么在这课里我将要为您展现如何实现这项功能。这是我第一次写教程，而且相对来说我也是 OpenGL/C++ 程序设计新手，所以如果您发现有什么错误的话，请让我知道。这课的代码是基于第七课的。

NeHeWidget 类

（由 nehewidget.h 展开。）

```
class NeHeWidget : public QGLWidget
{
    Q_OBJECT

public:

  NeHeWidget( QWidget* parent = 0, const char* name = 0, bool fs = false );
  ~NeHeWidget();
```

```
protected:

  void initializeGL();
  void paintGL();
  void resizeGL( int width, int height );

  void keyPressEvent( QKeyEvent *e );
  void loadGLTextures();

protected:

  bool fullscreen;

  GLfloat xRot, yRot, zRot;
  GLfloat zoom;
  GLfloat xSpeed, ySpeed;
  GLuint texture[3];
  GLuint filter;

  bool light;

  GLuint fogFilter;

};
```

变量 fogfilter，将被用做记录您所选择的雾的类型的索引。

（由 nehewidget.cpp 展开。）

```
GLfloat lightAmbient[4] = { 0.5, 0.5, 0.5, 1.0 };
GLfloat lightDiffuse[4] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat lightPosition[4] = { 0.0, 0.0, 2.0, 1.0 };

GLuint fogMode[3] = { GL_EXP, GL_EXP2, GL_LINEAR };
GLfloat fogColor[4] = { 0.5, 0.5, 0.5, 1.0 };
```

数据设定

我们将要设定我们用来保存关于雾的信息的所有变量。变量 fogMode，用来保存 3 种有关雾的类型：GL_EXP，GL_EXP2，GL_LINEAR。稍后我会解释这三种类型间的差别。这个变量将在代码的开头声明。变量 fogColor 会保存任何您想要的雾的颜色。

```
NeHeWidget::NeHeWidget( QWidget* parent, const char* name, bool fs )
    : QGLWidget( parent, name )
```

```
{
  xRot = yRot = zRot = 0.0;
  zoom = -5.0;
  xSpeed = ySpeed = 0.0;

  filter = 0;

  light = false;

  fogFilter = 0;

  fullscreen = fs;
  setGeometry( 0, 0, 640, 480 );
  setCaption( "Chris Aliotta & NeHe's Fog Tutorial" );

  if ( fullscreen )
    showFullScreen();
}
```

我们需要在构造函数中给各个变量赋初值。

```
void NeHeWidget::loadGLTextures()
{
  QImage tex, buf;
  if ( !buf.load( "./data/Crate.bmp" ) )
  {
    qWarning( "Could not read image file, using single-color instead." );
    QImage dummy( 128, 128, 32 );
    dummy.fill( Qt::green.rgb() );
    buf = dummy;
  }
  tex = QGLWidget::convertToGLFormat( buf );

  glGenTextures( 3, &texture[0] );

  glBindTexture( GL_TEXTURE_2D, texture[0] );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST );
  glTexImage2D( GL_TEXTURE_2D, 0, 3, tex.width(), tex.height(), 0,
      GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() );

  glBindTexture( GL_TEXTURE_2D, texture[1] );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR );
```

```
  glTexImage2D( GL_TEXTURE_2D, 0, 3, tex.width(), tex.height(), 0,
      GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() );

  glBindTexture( GL_TEXTURE_2D, texture[2] );
  glTexParameteri( GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR );
  gluBuild2DMipmaps( GL_TEXTURE_2D, GL_RGB, tex.width(), tex.height(),
      GL_RGBA, GL_UNSIGNED_BYTE, tex.bits() );

}
```

loadGLTextures()函数就是用来载入纹理的。

```
void NeHeWidget::initializeGL()
{
  loadGLTextures();

  glEnable( GL_TEXTURE_2D );
  glShadeModel( GL_SMOOTH );
  glClearColor( 0.5, 0.5, 0.5, 1.0 );
```

场景绘制设定

上面我们进行初始化 OpenGL 了。glClearColor()将会稍做改变以用来将屏幕颜色清除为雾的颜色以获得更好的视觉效果。这里并没有复杂的代码用来进行雾的操作，而且您会发现，这将是非常简单的。

```
  glClearDepth( 1.0 );
  glEnable( GL_DEPTH_TEST );
  glDepthFunc( GL_LEQUAL );
  glHint( GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST );

  glLightfv( GL_LIGHT1, GL_AMBIENT, lightAmbient );
  glLightfv( GL_LIGHT1, GL_DIFFUSE, lightDiffuse );
  glLightfv( GL_LIGHT1, GL_POSITION, lightPosition );

  glEnable( GL_LIGHT1 );

  glFogi( GL_FOG_MODE, fogMode[fogFilter] );
```

确定了雾的类型。开始的时候我们声明了数组 fogMode，它保存了 GL_EXP，GL_EXP2，GL_LINEAR。在我们使用这些值之前，让我稍做解释：

GL_EXP：简单渲染在屏幕上显示的雾的模式。它无法给予我们非常漂亮的雾的效果，但是却可以在古老的电脑上工作的很好。

GL_EXP2：比 1 提高了一点，将渲染全屏幕的雾，然而她会给予场景更深的效果。

GL_LINEAR：这是最好的雾的渲染模式，对象在雾中消隐的很好。

```
  glFogfv( GL_FOG_COLOR, fogColor );
```

设定了雾的颜色，开始的时候我们把它设定为(0.5,0.5,0.5,1.0)，使用变量 fogcolor 将给予我们漂亮的灰色。

```
  glFogf( GL_FOG_DENSITY, 0.35 );
```

确定了雾的密度，增大这个数值雾将会变的更浓，减小它雾将会变的更淡。

```
  glHint( GL_FOG_HINT, GL_DONT_CARE );
```

确定了雾的渲染方式，我使用 GL_DONT_CARE 是因为我并不关心建议值。然而这里有一个用来解释关于这个项的不同值之间的区别：

GK_DONT_CARE：让 OPENGL 自己来确定雾的渲染方式，每顶点或是每像素。

GL_NICEST：对每一像素进行雾的渲染，它看起来是极棒的。

GL_FASTEST：对每一顶点进行雾的渲染，它速度较快，但是不够美丽。

```
  glFogf( GL_FOG_START, 1.0 );
```

确定了雾的开始初离屏幕有多近。你可以将这个值改变为任意你想要的值，这个值描述了那个你想要使雾开始的位置。下一行与上行相似，它告诉 OpenGL 雾能离开屏幕有多远。

```
  glFogf( GL_FOG_END, 5.0 );
  glEnable( GL_FOG );
```

glEnable(GL_FOG)解释起来非常容易，它告诉 OpenGL 开始进行雾的计算。

```
}
void NeHeWidget::paintGL()
{
  glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT );
  glLoadIdentity();
  glTranslatef(  0.0,  0.0, zoom );

  glRotatef( xRot,  1.0,  0.0,  0.0 );
  glRotatef( yRot,  0.0,  1.0,  0.0 );
```

```
  glBindTexture( GL_TEXTURE_2D, texture[filter] );

  glBegin( GL_QUADS );
    glNormal3f(  0.0,  0.0, 1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0,  1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0,  1.0 );

    glNormal3f(  0.0,  0.0, -1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0, -1.0 );

    glNormal3f(  0.0,  1.0,  0.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0,  1.0,  1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0,  1.0,  1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );

    glNormal3f(  0.0, -1.0,  0.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0, -1.0, -1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );

    glNormal3f(  1.0,  0.0,  0.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f(  1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f(  1.0,  1.0, -1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f(  1.0,  1.0,  1.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f(  1.0, -1.0,  1.0 );

    glNormal3f( -1.0,  0.0,  0.0 );
    glTexCoord2f( 0.0, 0.0 ); glVertex3f( -1.0, -1.0, -1.0 );
    glTexCoord2f( 1.0, 0.0 ); glVertex3f( -1.0, -1.0,  1.0 );
    glTexCoord2f( 1.0, 1.0 ); glVertex3f( -1.0,  1.0,  1.0 );
    glTexCoord2f( 0.0, 1.0 ); glVertex3f( -1.0,  1.0, -1.0 );
  glEnd();

  xRot += xSpeed;
  yRot += ySpeed;
}
void NeHeWidget::keyPressEvent( QKeyEvent *e )
```

```
{
  switch ( e->key() )
  {
  case Qt::Key_L:
    light = !light;
    if ( !light )
    {
      glDisable( GL_LIGHTING );
    }
    else
    {
      glEnable( GL_LIGHTING );
    }
    updateGL();
    break;
  case Qt::Key_F:
    filter += 1;;
    if ( filter > 2 )
    {
      filter = 0;
    }
    updateGL();
    break;
  case Qt::Key_G:
    fogFilter += 1;;
    if ( fogFilter > 2 )
    {
      fogFilter = 0;
    }
    glFogi( GL_FOG_MODE, fogMode[fogFilter] );
    updateGL();
    break;
```

按下了 G 键，就可以变换一下所使用的雾的类型。

```
  case Qt::Key_Prior:
    zoom -= 0.2;
    updateGL();
    break;
  case Qt::Key_Next:
    zoom += 0.2;
    updateGL();
    break;
  case Qt::Key_Up:
```

```
    xSpeed -= 0.01;
    updateGL();
    break;
  case Qt::Key_Down:
    xSpeed += 0.01;
    updateGL();
    break;
  case Qt::Key_Right:
    ySpeed += 0.01;
    updateGL();
    break;
  case Qt::Key_Left:
    ySpeed -= 0.01;
    updateGL();
    break;
  case Qt::Key_F2:
    fullscreen = !fullscreen;
    if ( fullscreen )
    {
      showFullScreen();
    }
    else
    {
      showNormal();
      setGeometry( 0, 0, 640, 480 );
    }
    update();
    break;
  case Qt::Key_Escape:
    close();
  }
}
```

上面就是键盘控制，用上下左右键来控制立方体的运动。

这就是全部了！我们结束了我们的课程，现在你已经在你的 OPENGL 程序中有了雾。我敢说这是决不费事的。

本课程的源代码。