

# 课程索引

在这个课程上介绍的知识,可能会有错误和问题。但我只想尽我的全力去制作一个针对初学者的课程 , 让他们在学习感到轻松和简单。如果你对于学习OpenGL很认真 , 我建议你购买OpenGL Red Book(ISBN 0-201-46138-2)和OpenGL Blue Book , 它们是至今为止最好的参考书。另一本我要推荐的书是OpenGL Superbible,当然你也可以选择其它的。当你在这个网站上学习时 , 你应该多浏览一下其它的站点 , 如OpenGL.org. 你也可以浏览一些我提供的链接 , 大部分站点都是由一些非常优秀的编程人员创建的 , 它们很多都比我出色。我希望你能喜欢我提供给你的帮助 , 并且希望看到在不久的将来你能创建出你自己的程序。



## 创建一个OpenGL窗口:

在这个教程里,我将教你在Windows环境中创建OpenGL程序.它将显示一个空的OpenGL窗口,可以在窗口和全屏模式下切换,按ESC退出.它是以后应用程序的框架.

理解OpenGL如何工作非常重要 , 你可以在教程的末尾下载源程序 , 但我强烈建议你至少读一遍教程 , 然后再开始编程.



### 你的第一个多边形:

在第一个教程的基础上，我们添加了一个三角形和一个四边形。也许你认为这很简单，但你已经迈出了一大步，要知道任何在OpenGL中绘制的模型都会被分解为这两种简单的图形。

读完了这一课，你会学到如何在空间放置模型，并且会知道深度缓存的概念。



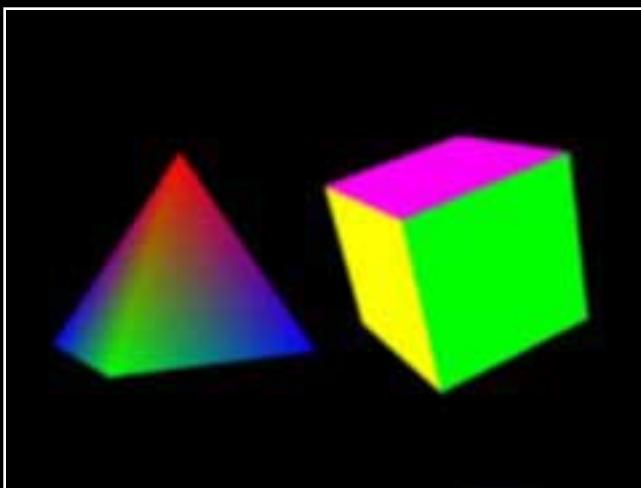
### 添加颜色:

作为第二课的扩展，我将叫你如何使用颜色。你将理解两种着色模式，在左图中，三角形用的是光滑着色，四边形用的是平面着色。



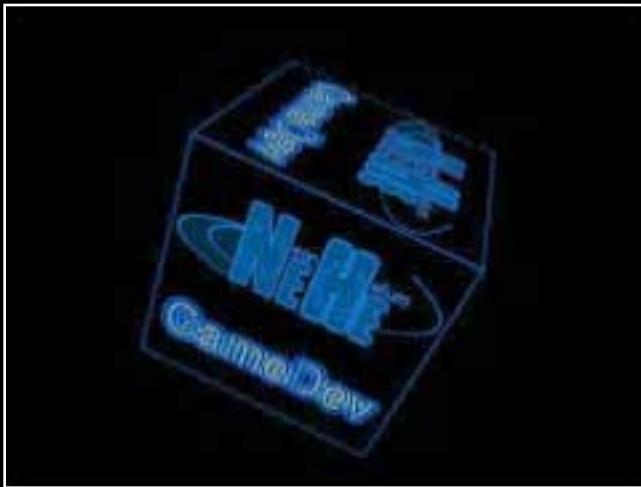
### 旋转:

在这一课里，我将教会你如何旋转三角形和四边形。左图中的三角形沿Y轴旋转，四边形沿着X轴旋转。



### 3D空间:

我们使用多边形和四边形创建3D物体，在这一课里，我们把三角形变为立体的金子塔形状，把四边形变为立方体。



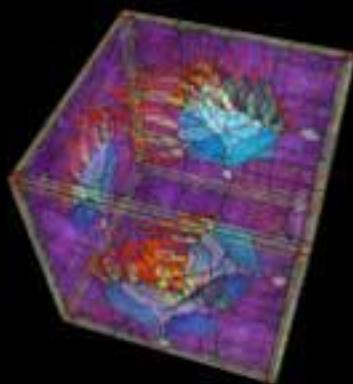
### 纹理映射:

在这一课里，我将教会你如何把纹理映射到立方体的六个面。



### 光照和键盘控制:

在这一课里，我们将添加光照和键盘控制，它让程序看起来更美观。



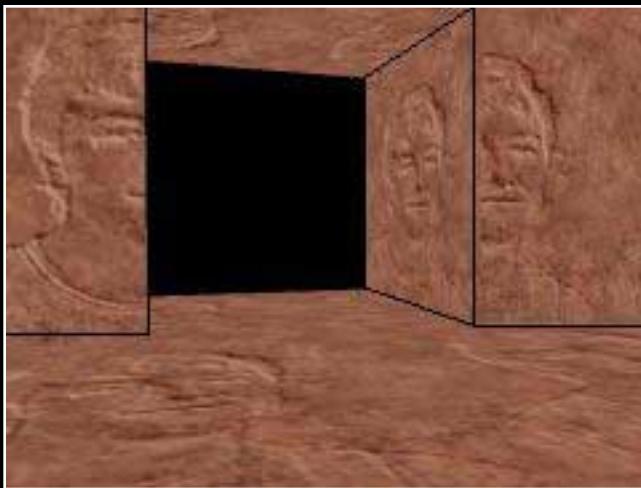
### 混合:

在这一课里，我们在纹理的基础上加上了混合，它看起具有透明的效果，当然解释它不是那么容易，当希望你喜欢它。



### 3D空间中移动图像:

你想知道如何在3D空间中移动物体，你想知道如何在屏幕上绘制一个图像，而让图像的背景色变为透明，你希望有一个简单的动画。这一课将教会你所有的一切。前面的课程涵盖了基础的OpenGL，每一课都是在前一课的基础上创建的。这一课是前面几课知识的综合，当你学习这课时，请确保你已经掌握了前面几课的知识。



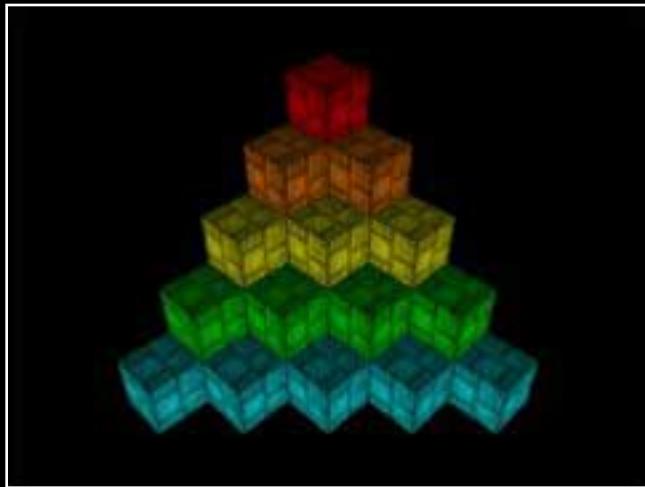
### 加载3D世界，并在其中漫游:

在这一课中，你将学会如何加载3D世界，并在3D世界中漫游。这一课使用第一课的代码，当然在课程说明中我只介绍改变了代码。



## 飘动的旗帜:

这一课从第六课的代码开始，创建一个飘动的旗帜。我相信在这课结束的时候，你可以掌握纹理映射和混合操作。



## 显示列表:

想知道如何加速你的OpenGL程序么？这一课将告诉你如何使用OpenGL的显示列表，它通过预编译OpenGL命令来加速你的程序，并可以为你省去很多重复的代码。



## 图像字体:

这一课我们将创建一些基于2D图像的字体，它们可以缩放，但不能旋转，并且总是面向前方，但作为基本的显示来说，我想已经够了。



### 图形字体:

在一课我们将教你绘制3D的图形字体，它们可像一般的3D模型一样被变换。



### 图形字体的纹理映射:

这一课，我们将在上一课的基础上创建带有纹理的字体，它真的很简单。



### 雾:

这一课是基于第7课的代码的，你将学会三种不同的雾的计算方法，以及怎样设置雾的颜色和雾的范围。



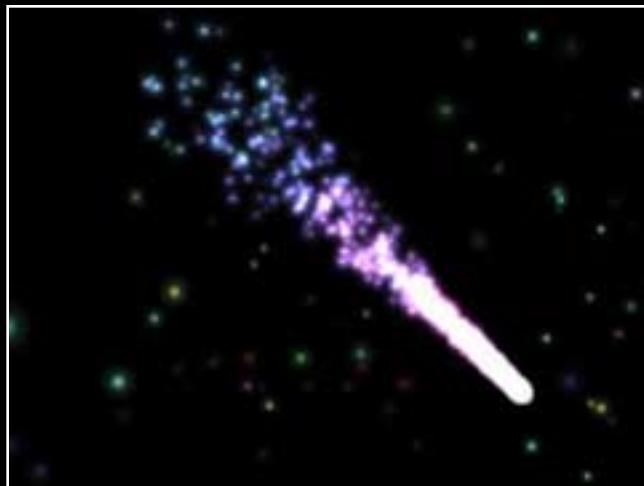
### 2D 图像文字:

在这一课中，你将学会如何使用四边形纹理贴图把文字显示在屏幕上。你将学会如何把256个不同的文字从一个256x256的纹理图像中分别提取出来，并为每一个文字创建一个显示列表，接着创建一个输出函数来创建任意你希望的文字。



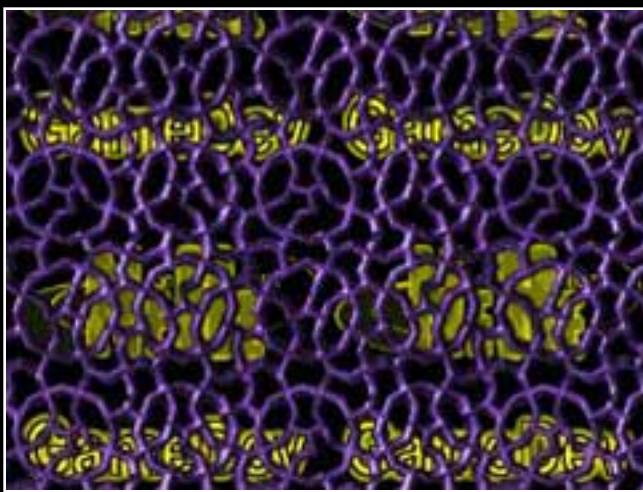
### 二次几何体:

利用二次几何体，你可以很容易的创建球，圆盘，圆柱和圆锥。



### 粒子系统:

你是否希望创建爆炸，喷泉，流星之类的效果。这一课将告诉你如何创建一个简单的例子系统，并用它来创建一种喷射的效果。



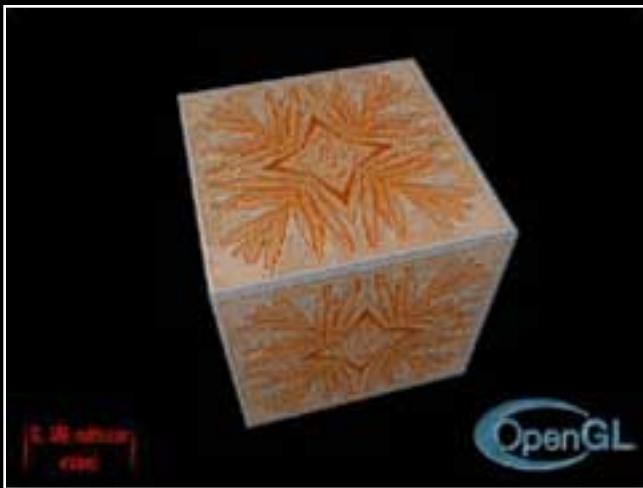
## 蒙板:

到目前为止你已经学会如何使用alpha混合，把一个透明物体渲染到屏幕上，但有的使用它看起来并不是那么的复合你的心意。使用蒙板技术，将会按照你蒙板的位置精确的绘制。



## 线，反走样，计时，正投影和简单的声音:

这是我第一个大的教程，它将包括线，反走样，计时，正投影和简单的声音。希望这一课中的东西能让每个人感到高兴。



## 凹凸映射，多重纹理扩展:

这是一课高级教程，请确信你对基本知识已经非常了解了。这一课是基于第六课的代码的，它将建立一个非常酷的立体纹理效果。





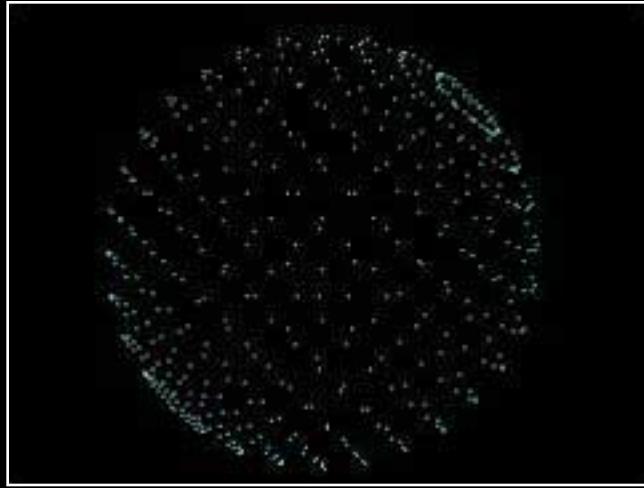
## 球面映射:

这一个将教会你如何把环境纹理包裹在你的3D模型上，让它看起来象反射了周围的场景一样。



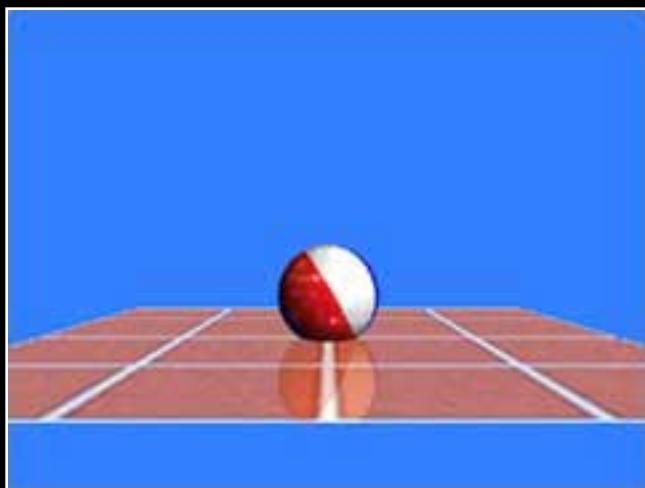
## 扩展，剪裁和TGA图像文件的加载:

在这一课里，你将学会如何读取你显卡支持的OpenGL的扩展，并在你指定的剪裁区域把它显示出来。



## 变形和从文件中加载3D物体:

在这一课中，你将学会如何从文件加载3D模型，并且平滑的从一个模型变换为另一个模型。



## 剪裁平面，蒙板缓存和反射:

在这一课中你将学会如何创建镜面显示效果，它使用剪裁平面，蒙板缓存等OpenGL中一些高级的技巧。



## 影子:

这是一个高级的主题，请确信你已经熟练的掌握了基本的OpenGL，并熟悉蒙板缓存。当然它会给你留下深刻的印象的。



## 贝塞尔曲面:

这是一课关于数学运算的，没有别的内容了。来，有信心就看看它吧。





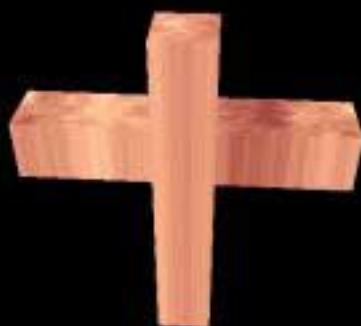
### Blitter 函数:

类似于DirectDraw的blit函数，过时的技术，我们有实现了它。它非常的简单，就是把一块纹理贴到另一块纹理上。



### 碰撞检测:

这是一课激动的教程，你也许等待它多时了。你将学会碰撞剪裁，物理模拟太多的东西，慢慢期待吧。



### 模型加载:

你知道大名鼎鼎的Milkshape3D建模软件么，我们将加载它的模型，当然你可以加载任何你认为不错的模型。



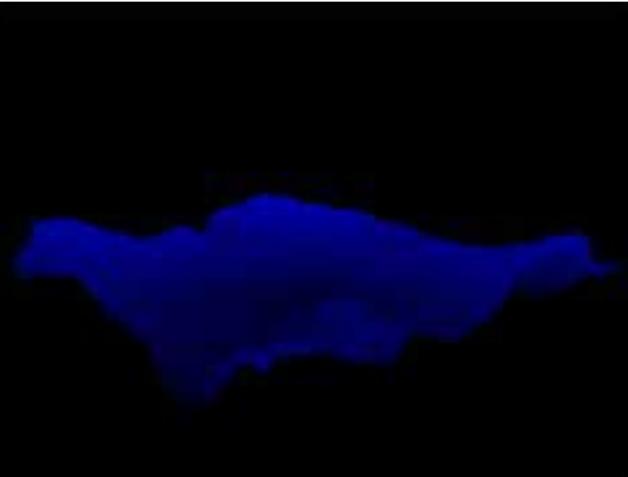
### 拾取, Alpha混合, Alpha测试, 排序:

这又是一个小游戏，交给的东西会很多，慢慢体会吧



### 加载压缩和未压缩的TGA文件:

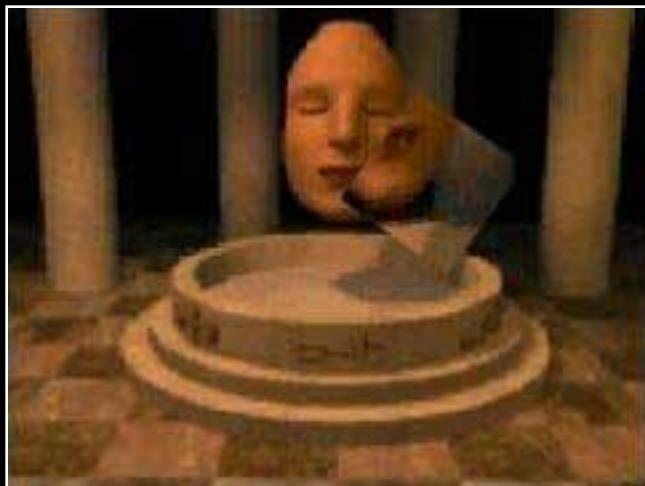
在这一课里，你将学会如何加载压缩和为压缩的TGA文件，由于它使用RLE压缩，所以非常的简单，你能很快地熟悉它的。



### 从高度图生成地形:

这一课将教会你如何从一个2D的灰度图创建地形





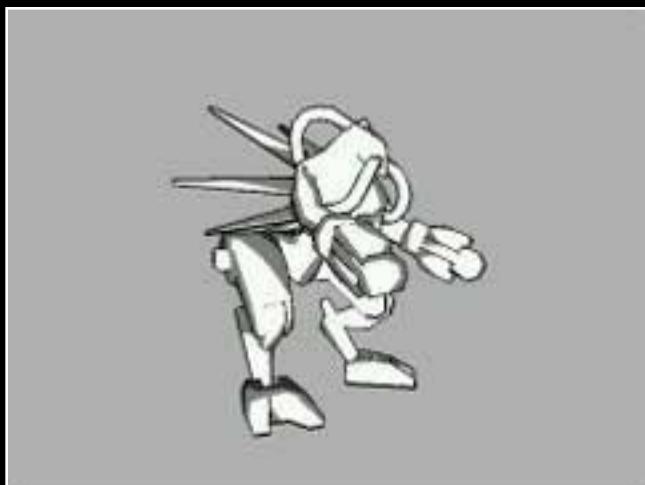
### 在OpenGL中播放AVI:

在OpenGL中如何播放AVI呢？利用Windows的API把每一帧作为纹理绑定到OpenGL中，虽然很慢，但它的效果不错。你可以试试。



### 放射模糊和渲染到纹理:

如何实现放射状的滤镜效果呢，看上去很难，其实很简单。把渲染得图像作为纹理提取出来，在利用OpenGL本身自带的纹理过滤，就能实现这种效果，不信，你试试。



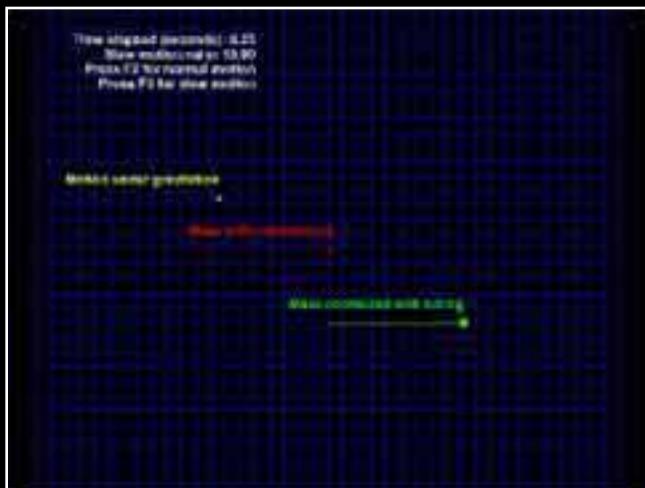
### 卡通映射:

什么是卡通了，一个轮廓加上少量的几种颜色。使用一维纹理映射，你也可以实现这种效果。



### 从资源文件中载入图像:

如何把图像数据保存到\*.exe程序中，使用Windows的资源文件吧，它既简单又实用。



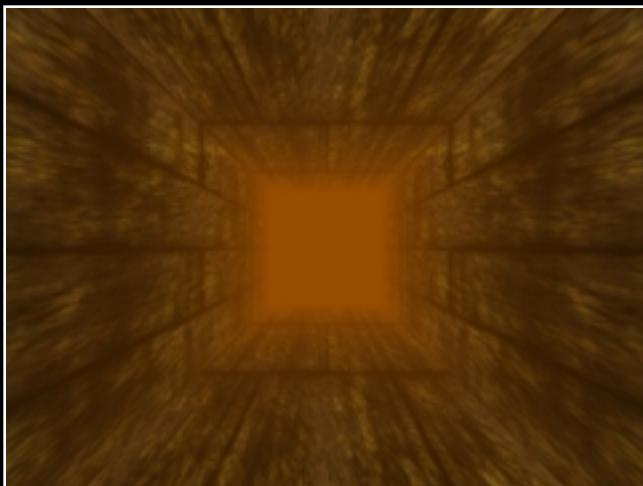
### 物理模拟简介:

还记得高中的物理吧，直线运动，自由落体运动，弹簧。在这一课里，我们将创造这一切。



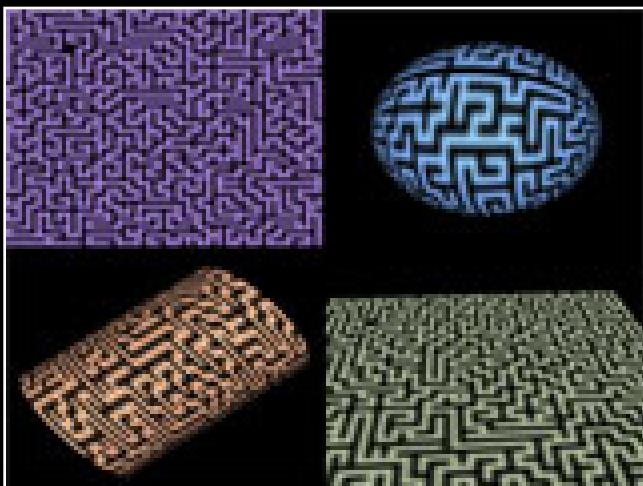
### 绳子的模拟:

怎样模拟一根绳子呢，把它想象成一个个紧密排列的点，怎么样有了思路了吧，在这一课你将学会怎样建模，简单吧，你能模拟更多。



### 体积雾气

把雾坐标绑定到顶点，你可以在雾中漫游，体验一下吧。



### 多重视口

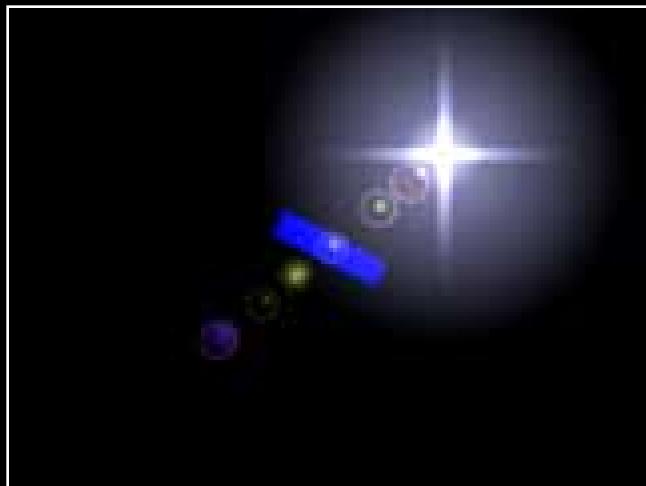
画中画效果，很酷吧。使用视口它变得很简单，但渲染四次会大大降低你的显示速度哦：）

### Active WGL Bitmap Text With NeHe - 311.75

Active FreeType Text - 311.75

### 在OpenGL中使用FreeType库

使用FreeType库可以创建非常好看的反走样的字体，记住暴雪公司就是使用这个库的，就是那个做魔兽世界的。尝试一下吧，我只告诉你了基本的使用方式，你可以走的更远。



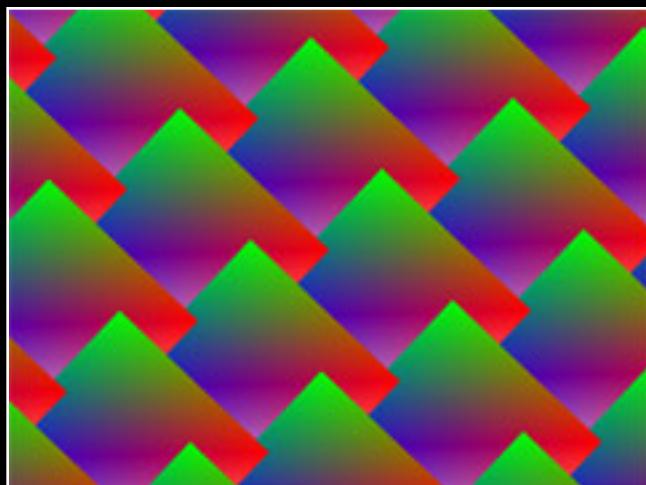
### 3D 光晕

当镜头对准太阳的时候就会出现这种效果，模拟它非常的简单，一点数学和纹理贴图就够了。好好看看吧。



### 顶点缓存

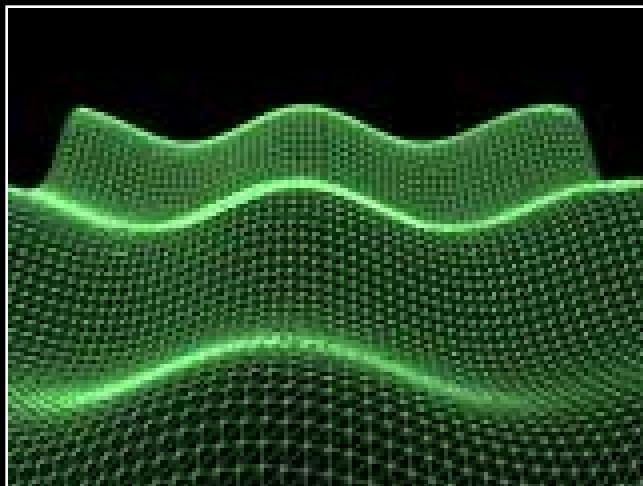
你想更快地绘制么？直接操作显卡吧，这可是当前的图形技术，不要犹豫，我带你入门。接下来，你自己向前走吧。



### 全屏反走样

当今显卡的强大功能，你几乎什么都不用做，只需要在创建窗口的时候该一个数据。看看吧，驱动程序为你做完了一切。





### CG 顶点脚本

nVidia的面向GPU的C语言，如果你相信它就好好学学吧，同样这里也只是个入门。记住，类似的语言还有微软的HLSL,OpenGL的GLSL,ATI的shaderMonker。不要选错哦:)



### 轨迹球实现的鼠标旋转

使用鼠标旋转物体,很简单也有很多实现方法,这里我们教会你模拟轨迹球来实现它.

我不是一个天才的程序员，我只是一个普通的编程者，每一天都在学习OpenGL。我不宣称我了解OpenGL的每一个方面，也不保证我的代码没有bug，但我会尽一切可能去剔除我的程序中的bugs。

当你在学习上面的课程时，请记住我的话。

### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

### 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。





## 第02课

你的第一个多边形：



在第一个教程的基础上，我们添加了一个三角形和一个四边形。也许你认为这很简单，但你已经迈出了一大步，要知道任何在OpenGL中绘制的模型都会被分解为这两种简单的图形。

读完了这一课，你会学到如何在空间放置模型，并且会知道深度缓存的概念。

第一课中，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。我们讲使用来创建GL\_TRIANGLES一个三角形，GL\_QUADS来创建一个四边形。

在第一课代码的基础上，我们只需在DrawGLScene()过程中增加代码。下面我重写整个过程。如果您计划修改上节课的代码，只需用下面的代码覆盖原来的DrawGLScene()就可以了。

```
int DrawGLScene(GLvoid) // 此过程中包括所有的绘制代码
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除屏幕及深度缓存
    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0.0f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(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(0.0f)，最后移入屏幕6.0f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。

```
glTranslatef(-1.5f,0.0f,-6.0f); // 左移 1.5 单位，并移入屏幕 6.0
```

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)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。

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。

```
glBegin(GL_TRIANGLES); // 绘制三角形
    glVertex3f( 0.0f, 1.0f, 0.0f); // 上顶点
    glVertex3f(-1.0f,-1.0f, 0.0f); // 左下
    glVertex3f( 1.0f,-1.0f, 0.0f); // 右下
glEnd(); // 三角形绘制结束
```

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。

```
glTranslatef(3.0f,0.0f,0.0f); // 右移3单位
```

现在使用GL\_QUADS绘制正方形。与绘制三角形的代码相类似，画四边形也很简单。唯一的区别是用GL\_QUADS来替换了GL\_TRIANGLES。并增加了一个点。我们使用顺时针次序来画正方形 - 左上 - 右上 - 右下 - 左下。采用顺时针绘制的是对象的后表面。这就是说我们所看见的是正方形的背面。逆时针画出来的正方形才是正面朝着我们的。现在这对您来说并不重要，但以后您必须知道。

```
glBegin(GL_QUADS); // 绘制正方形
    glVertex3f(-1.0f, 1.0f, 0.0f); // 左上
    glVertex3f( 1.0f, 1.0f, 0.0f); // 右上
    glVertex3f( 1.0f,-1.0f, 0.0f); // 左下
    glVertex3f(-1.0f,-1.0f, 0.0f); // 右下
glEnd(); // 正方形绘制结束
return TRUE; // 继续运行
}
```

最后换掉窗口模式下的标题内容。

```
if (keys[VK_F1]) // F1键按下了么?
{
    keys[VK_F1]=FALSE; // 若是，使对应的Key数组中的值为 FALSE
    KillGLWindow(); // 销毁当前的窗口
    fullscreen=!fullscreen; // 切换 全屏 / 窗口 模式
    // 重建 OpenGL 窗口(修改)
    if (!CreateGLWindow("NeHe's 第一个多边形程序",640,480,16,fullscreen))
    {
        return 0; // 如果窗口未能创建，程序退出
    }
}
```

{  
}

在这一课中，我已试着尽量详细的解释与多边形绘制有关的步骤。并创建了一个绘制三角形和正方形的OpenGL程序。如果您有什么意见或建议请给我EMAIL。如果您认为有什么不对或可以改进，请告诉我。我想做最好的OpenGL教程并对您的反馈感兴趣。

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

#### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

#### 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

#### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照



顾。

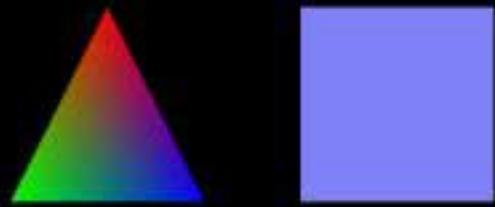
感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第01课

第03课 >



## 第03课



添加颜色:

作为第二课的扩展，我将叫你如何使用颜色。你将理解两种着色模式，在左图中，三角形用的是光滑着色，四边形用的是平面着色。

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2种不同类型的着色方法。使用Flat coloring(单调着色)给四边形涂上固定的一种颜色。使用Smooth coloring(平滑着色)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。

继续在上节课的DrawGLScene例程上修改。下面将整个例程重写了一遍。如果您计划修改上节课的代码，只需用下面的代码覆盖原来的DrawGLScene()就可以了。

```

int DrawGLScene(GLvoid) // 此过程中包括所有的绘制代码
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除屏幕及深度缓存
    glLoadIdentity(); // 重置模型观察矩阵
    glTranslatef(-1.5f,0.0f,-6.0f); // 左移 1.5 单位，并移入屏幕 6.0

    glBegin(GL_TRIANGLES); // 绘制三角形

```

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。下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0.0f到1.0f。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)。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(1.0f,0.0f,0.0f);           // 设置当前色为红色  
glVertex3f( 0.0f, 1.0f, 0.0f);       // 上顶点
```

第一个红色顶点已经设置完毕。接下来我们设置第二个绿色顶点。三角形的左下顶点被设为绿色。

```
glColor3f(0.0f,1.0f,0.0f);           // 设置当前色为绿色  
glVertex3f(-1.0f,-1.0f, 0.0f);      // 左下
```

现在设置第三个也就是最后一个顶点。开始绘制之前将颜色设为蓝色。这将是三角形的右下顶点。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。

```
glColor3f(0.0f,0.0f,1.0f);           // 设置当前色为蓝色  
glVertex3f( 1.0f,-1.0f, 0.0f);       // 右下  
glEnd();                            // 三角形绘制结束  
  
glTranslatef(3.0f,0.0f,0.0f);         // 右移3单位
```

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。

我们必须做的事情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。

```
glColor3f(0.5f,0.5f,1.0f);           // 一次性将当前色设置为蓝色
glBegin(GL_QUADS);                  // 绘制正方形
    glVertex3f(-1.0f, 1.0f, 0.0f);    // 左上
    glVertex3f( 1.0f, 1.0f, 0.0f);    // 右上
    glVertex3f( 1.0f,-1.0f, 0.0f);    // 左下
    glVertex3f(-1.0f,-1.0f, 0.0f);    // 右下
glEnd();                            // 正方形绘制结束
return TRUE;                         // 继续运行
}
```

最后换掉窗口模式下的标题内容

```
// 重建 OpenGL 窗口
if (!CreateGLWindow("NeHe's颜色实例",640,480,16,fullscreen))
```

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y有什么样的结果。如果您有什么意见或建议请给我EMAIL。如果您认为有什么不对或可以改进，请告诉我。我想做最好的OpenGL教程并对您的反馈感兴趣。

### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

### 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望

||我能带给她幸福。

< 第02课

第04课 >



## 第04课



旋转:

在这一课里，我将教会你如何旋转三角形和四边形。左图中的三角形沿Y轴旋转，四边形沿着X轴旋转。

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。下面我将整个例程重写一遍。方便您知道增加了什么，修改了什么。

我们增加两个变量来控制这两个对象的旋转。这两个变量加在程序的开始处其他变量的后面( `bool fullscreen=TRUE;` 下面的两行)。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。

```
GLfloat rtri; // 用于三角形的角度  
GLfloat rquad; // 用于四边形的角度
```

接着我们修改DrawGLScene()的代码。

下面这段代码与上一课的相同。

```
int DrawGLScene(GLvoid) // 此过程中包括所有的绘制代码
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除屏幕及深度缓存
    glLoadIdentity(); // 重置模型观察矩阵
    glTranslatef(-1.5f,0.0f,-6.0f); // 左移 1.5 单位，并移入屏幕 6.0
```

下一行代码是新的。glRotatef(Angle,Xvector,Yvector,Zvector)负责让对象绕某个轴旋转。这个命令有很多用处。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。(译者注：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。

下面的一行代码中，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。

```
glRotatef(rtri,0.0f,1.0f,0.0f); // 绕Y轴旋转三角形
```

下面的代码没有变化。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。

```
glBegin(GL_TRIANGLES);           // 绘制三角形
    glColor3f(1.0f,0.0f,0.0f);   // 设置当前色为红色
        glVertex3f( 0.0f, 1.0f, 0.0f); // 上顶点
    glColor3f(0.0f,1.0f,0.0f);   // 设置当前色为绿色
        glVertex3f(-1.0f,-1.0f, 0.0f); // 左下
    glColor3f(0.0f,0.0f,1.0f);   // 设置当前色为蓝色
        glVertex3f( 1.0f,-1.0f, 0.0f); // 右下
    glEnd();                     // 三角形绘制结束
```

您会注意下面的代码中我们增加了另一个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。

```
glLoadIdentity();           // 重置模型观察矩阵
glTranslatef(1.5f,0.0f,-6.0f); // 右移1.5单位，并移入屏幕 6.0
glRotatef(rquad,1.0f,0.0f,0.0f); // 绕X轴旋转四边形
```

下一段代码保持不变。在屏幕的右侧画一个蓝色的正方形

```
glColor3f(0.5f,0.5f,1.0f); // 一次性将当前色设置为蓝色
glBegin(GL_QUADS);         // 绘制正方形
    glVertex3f(-1.0f, 1.0f, 0.0f); // 左上
    glVertex3f( 1.0f, 1.0f, 0.0f); // 右上
```

```

glVertex3f( 1.0f,-1.0f, 0.0f);           // 左下
glVertex3f(-1.0f,-1.0f, 0.0f);           // 右下
glEnd();                                // 正方形绘制结束

```

下两行是新增的。倘若把 rtri 和 rquad 想象为容器，那么在程序的开始我们创建了容器 ( GLfloat rtri , 和 GLfloat rquad )。当容器创建之后，里面是空的。下面的第一行代码是向容器中添加0.2。因此每次当我们运行完前面的代码后，都会在这里使 rtri 容器中的值增长0.2。后面一行将 rquad 容器中的值减少0.15。同样每次当我们运行完前面的代码后，都会在这里使 rquad 容器中的值下跌0.15。下跌最终会导致对象旋转的方向和增长的方向相反。

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。

```

rtri+=0.2f;                           // 增加三角形的旋转变量
rquad-=0.15f;                          // 减少四边形的旋转变量
return TRUE;                           // 继续运行
}

```

最后换掉窗口模式下的标题内容

```

// 重建 OpenGL 窗口
if (!CreateGLWindow("NeHe's 旋转实例",640,480,16,fullscreen))

```

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:)。如果您有什么意见或建议请给我EMAIL。如果您认为有什么不对或可以改进，请告诉我。我想做最好的OpenGL教程并对您的反馈感兴趣。



### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

### 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

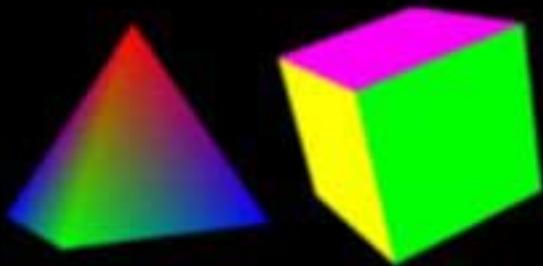
感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。

< 第03课

第05课 >



## 第05课



3D空间:

我们使用多边形和四边形创建3D物体，在这一课里，我们把三角形变为立体的金子塔形状，把四边形变为立方体。

在上节课的内容上作些扩展，我们现在开始生成真正的3D对象，而不是象前两节课中那样3D世界中的2D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。

```
int DrawGLScene(GLvoid) // 此过程中包括所有的绘制代码
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除屏幕及深度缓存
    glLoadIdentity(); // 重置模型观察矩阵
    glTranslatef(-1.5f,0.0f,-6.0f); // 左移 1.5 单位，并移入屏幕 6.0
    glRotatef(rtri,0.0f,1.0f,0.0f); // 绕Y轴旋转金字塔
```

```
glBegin(GL_TRIANGLES);
```

// 开始绘制金字塔的各个面

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(0.0f,0.0f,0.0f)。

下面的代码创建一个绕着其中心轴旋转的金字塔。金字塔的上顶点高出原点一个单位，底面中心低于原点一个单位。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。

我们开始画金字塔的前侧面。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。

```
glColor3f(1.0f,0.0f,0.0f);           // 红色
glVertex3f( 0.0f, 1.0f, 0.0f);       // 三角形的上顶点 (前侧面)
                                     glColor3f(0.0f,1.0f,0.0f);       // 绿色
glVertex3f(-1.0f,-1.0f, 1.0f);       // 三角形的左下顶点 (前侧面)
                                     glColor3f(0.0f,0.0f,1.0f);       // 蓝色
glVertex3f( 1.0f,-1.0f, 1.0f);       // 三角形的右下顶点 (前侧面)
```

现在绘制右侧面。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。

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(1.0f,0.0f,0.0f);           // 红色
```

```

glVertex3f( 0.0f, 1.0f, 0.0f);           // 三角形的上顶点 (右侧面)
glColor3f(0.0f,0.0f,1.0f);           // 蓝色
glVertex3f( 1.0f,-1.0f, 1.0f);         // 三角形的左下顶点 (右侧面)
glColor3f(0.0f,1.0f,0.0f);           // 绿色
glVertex3f( 1.0f,-1.0f, -1.0f);        // 三角形的右下顶点 (右侧面)

```

现在是后侧面。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(1.0f,0.0f,0.0f);           // 红色
glVertex3f( 0.0f, 1.0f, 0.0f);         // 三角形的上顶点 (后侧面)
glColor3f(0.0f,1.0f,0.0f);           // 绿色
glVertex3f( 1.0f,-1.0f, -1.0f);        // 三角形的左下顶点 (后侧面)
glColor3f(0.0f,0.0f,1.0f);           // 蓝色
glVertex3f(-1.0f,-1.0f, -1.0f);        // 三角形的右下顶点 (后侧面)

```

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。

```

glColor3f(1.0f,0.0f,0.0f);           // 红色
glVertex3f( 0.0f, 1.0f, 0.0f);         // 三角形的上顶点 (左侧面)
glColor3f(0.0f,0.0f,1.0f);           // 蓝色
glVertex3f(-1.0f,-1.0f, -1.0f);        // 三角形的左下顶点 (左侧面)
glColor3f(0.0f,1.0f,0.0f);           // 绿色
glVertex3f(-1.0f,-1.0f, 1.0f);        // 三角形的右下顶点 (左侧面)
glEnd();                            // 金字塔绘制结束

```

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)。

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:)。

```
glLoadIdentity();
glTranslatef(1.5f,0.0f,-7.0f);           // 先右移再移入屏幕

glRotatef(rquad,1.0f,1.0f,1.0f);        // 在XYZ轴上旋转立方体

glBegin(GL_QUADS);                     // 开始绘制立方体
```

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(0.0f,1.0f,0.0f);           // 颜色改为蓝色
	glVertex3f( 1.0f, 1.0f,-1.0f);      // 四边形的右上顶点(顶面)
	glVertex3f(-1.0f, 1.0f,-1.0f);      // 四边形的左上顶点(顶面)
	glVertex3f(-1.0f, 1.0f, 1.0f);      // 四边形的左下顶点(顶面)
	glVertex3f( 1.0f, 1.0f, 1.0f);      // 四边形的右下顶点(顶面)
```

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(1.0f,0.5f,0.0f);           // 颜色改成橙色
	glVertex3f( 1.0f,-1.0f, 1.0f);      // 四边形的右上顶点(底面)
```

```

glVertex3f(-1.0f,-1.0f, 1.0f);           // 四边形的左上顶点(底面)
glVertex3f(-1.0f,-1.0f,-1.0f);          // 四边形的左下顶点(底面)
glVertex3f( 1.0f,-1.0f,-1.0f);          // 四边形的右下顶点(底面)

```

接着画立方体的前面。保持Z坐标为一单位，前面正对着我们。

```

glColor3f(1.0f,0.0f,0.0f);           // 颜色改成红色
glVertex3f( 1.0f, 1.0f, 1.0f);       // 四边形的右上顶点(前面)
glVertex3f(-1.0f, 1.0f, 1.0f);       // 四边形的左上顶点(前面)
glVertex3f(-1.0f,-1.0f, 1.0f);       // 四边形的左下顶点(前面)
glVertex3f( 1.0f,-1.0f, 1.0f);       // 四边形的右下顶点(前面)

```

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(1.0f,1.0f,0.0f);           // 颜色改成黄色
glVertex3f( 1.0f,-1.0f,-1.0f);      // 四边形的右上顶点(后面)
glVertex3f(-1.0f,-1.0f,-1.0f);      // 四边形的左上顶点(后面)
glVertex3f(-1.0f, 1.0f,-1.0f);      // 四边形的左下顶点(后面)
glVertex3f( 1.0f, 1.0f,-1.0f);      // 四边形的右下顶点(后面)

```

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(0.0f,0.0f,1.0f);           // 颜色改成蓝色
glVertex3f(-1.0f, 1.0f, 1.0f);      // 四边形的右上顶点(左面)
glVertex3f(-1.0f, 1.0f,-1.0f);      // 四边形的左上顶点(左面)
glVertex3f(-1.0f,-1.0f,-1.0f);      // 四边形的左下顶点(左面)
glVertex3f(-1.0f,-1.0f, 1.0f);      // 四边形的右下顶点(左面)

```

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：）

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。

```
glColor3f(1.0f,0.0f,1.0f);           // 颜色改成紫罗兰色
glVertex3f( 1.0f, 1.0f,-1.0f);       // 四边形的右上顶点(右面)
glVertex3f( 1.0f, 1.0f, 1.0f);       // 四边形的左上顶点(右面)
glVertex3f( 1.0f,-1.0f, 1.0f);       // 四边形的左下顶点(右面)
glVertex3f( 1.0f,-1.0f,-1.0f);      // 四边形的右下顶点(右面)
glEnd();                            // 立方体绘制结束

rtri+=0.2f;                         // 增加三角形的旋转变量
rquad-=0.15f;                        // 减少四边形的旋转变量
return TRUE;                          // 继续运行
}
```

这一课又结束了。到这里您应该已经较好的掌握了在3D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3D对象时没有任何问题。

如果您对3D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！如果您有什么意见或建议请给我EMAIL。如果您认为有什么不对或可以改进，请告诉我。我想做最好的OpenGL教程并对您的反馈感兴趣。



## 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

## 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

## 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

## 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

## 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。



## 第06课



纹理映射:

在这一课里，我将教会你如何把纹理映射到立方体的六个面。

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。

现在我们在第一课的代码开始处增加五行新代码。新增的第一行是 #include <stdio.h>。它允许我们对文件进行操作，为了在后面的代码中使用 fopen()，我们增加了这一行。然后我们增加了三个新的浮点变量... xrot , yrot 和 zrot。这些变量用来使立方体绕X、Y、Z轴旋转。最后一行 GLuint texture[1] 为一个纹理分配存储空间。如果您需要不止一个的纹理，应该将参数1改成您所需要的参数。

```
#include    <stdio.h>           // 标准输入/输出库的头文件
#include    <glaux.h>          // GLaux库的头文件

GLfloat    xrot;                // X 旋转量
```

```

GLfloat     yrot;           // Y 旋转量
GLfloat     zrot;           // Z 旋转量

GLuint      texture[1];     // 存储一个纹理

```

紧跟上面的代码在 ReSizeGLScene() 之前，我们增加了下面这一段代码。这段代码用来加载位图文件。如果文件不存在，返回 NULL 告知程序无法加载位图。在我开始解释这段代码之前，关于用作纹理的图像我想有几点十分重要，并且您必须明白。此图像的宽和高必须是2的n次方；宽度和高度最小必须是64象素；并且出于兼容性的原因，图像的宽度和高度不应超过256象素。如果您的原始素材的宽度和高度不是64,128,256象素的话，使用图像处理软件重新改变图像的大小。可以肯定有办法能绕过这些限制，但现在我们只需要用标准的纹理尺寸。

首先，我们创建一个文件句柄。句柄是个用来鉴别资源的数值，它使程序能够访问此资源。我们开始先将句柄设为 NULL。

```

AUX_RGBImageRec *LoadBMP(char *Filename)           // 载入位图图象
{
    FILE *File=NULL;                                // 文件句柄

```

接下来检查文件名是否已提供。因为 LoadBMP() 可以无参数调用，所以我们不得不检查一下。您可不想什么都没载入吧……：)

```

if (!Filename)                                     // 确保文件名已提供
{
    return NULL;                                    // 如果没提供，返回 NULL
}

```

接着检查文件是否存在。下面这一行尝试打开文件。

```

File=fopen(Filename,"r");                         // 尝试打开文件

```

如果我们能打开文件的话，很显然文件是存在的。使用 fclose(File) 关闭文件。  
auxDIBImageLoad(Filename) 读取图象数据并将其返回。

```
if (File) // 文件存在么?
{
    fclose(File); // 关闭句柄
    return auxDIBImageLoad(Filename); // 载入位图并返回指针
}
```

如果我们不能打开文件，我们将返回NULL。这意味着文件无法载入。程序在后面将检查文件是否已载入。如果没有，我们将退出程序并弹出错误消息。

```
return NULL; // 如果载入失败，返回 NULL
}
```

下一部分代码载入位图(调用上面的代码)并转换成纹理。

```
int LoadGLTextures() // 载入位图(调用上面的代码)并转换成纹理
{
```

然后设置一个叫做 Status 的变量。我们使用它来跟踪是否能够载入位图以及能否创建纹理。 Status 缺省设为 FALSE (表示没有载入或创建任何东东)。

```
int Status=FALSE; // 状态指示器
```

现在我们创建存储位图的图像记录。次记录包含位图的宽度、高度和数据。

```
AUX_RGBImageRec *TextureImage[1]; // 创建纹理的存储空间
```

清除图像记录，确保其内容为空

```
memset(TextureImage,0,sizeof(void *)*1); // 将指针设为 NULL
```

现在载入位图，并将其转换为纹理。 TextureImage[0]=LoadBMP("Data/NeHe.bmp") 调用 LoadBMP() 的代码。载入 Data 目录下的 NeHe.bmp 位图文件。如果一切正常，图像数据将存放在 TextureImage[0] 中， Status 被设为 TRUE ，然后我们开始创建纹理。

```
// 载入位图，检查有无错误，如果位图没找到则退出
if (TextureImage[0]=LoadBMP("Data/NeHe.bmp"))
{
    Status=TRUE; // 将 Status 设为 TRUE
```

现在使用中 TextureImage[0] 的数据创建纹理。第一行 glGenTextures(1, &texture[0]) 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。『译者注：学C的，在这里应该没有障碍，数组就是从零开始的嘛。』

第二行 glBindTexture(GL\_TEXTURE\_2D, texture[0]) 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。

```
glGenTextures(1, &texture[0]); // 创建纹理
```

```
// 使用来自位图数据生成 的典型纹理
glBindTexture(GL_TEXTURE_2D, texture[0]);
```

下来我们创建真正的纹理。下面一行告诉OpenGL此纹理是一个2D纹理(GL\_TEXTURE\_2D)。参数“0”代表图像的详细程度，通常就由它为零去了。参数三是数据的分数组。因为图像是由红色数据，绿色数据，蓝色数据三种组分组成。TextureImage[0]->sizeX是纹理的宽度。如果您知道宽度，您可以在那里填入，但计算机可以很容易的为您指出此值。TextureImage[0]->sizeY是纹理的高度。参数零是边框的值，一般就是“0”。GL\_RGB告诉OpenGL图像数据由红、绿、蓝三色数据组成。GL\_UNSIGNED\_BYTE意味着组成图像的数据是无符号字节类型的。最后... TextureImage[0]->data告诉OpenGL纹理数据的来源。此例中指向存放在TextureImage[0]记录中的数据。

```
// 生成纹理
glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[0]->sizeX, TextureImage[0]->sizeY, 0,
GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
```

下面的两行告诉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『译者注：马赛克啦』。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。

```
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR); // 线形滤波
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR); // 线形滤波
}
```

现在我们释放前面用来存放位图数据的内存。我们先查看位图数据是否存放在处。如果是的话，再查看数据是否已经存储。如果已经存储的话，删了它。接着再释放TextureImage[0]图像结构以保证所有的内存都能释放。

```

if (TextureImage[0]) // 纹理是否存在
{
    if (TextureImage[0]->data) // 纹理图像是否存在
    {
        free(TextureImage[0]->data); // 释放纹理图像占用的内存
    }
    free(TextureImage[0]); // 释放图像结构
}

```

最后返回状态变量。如果一切OK，变量 Status 的值为 TRUE。否则为 FALSE。

```

return Status; // 返回 Status
}

```

我只在 InitGL 中增加很少的几行代码。但为了方便您查看增加了哪几行，我这段代码全部重贴一遍。if (!LoadGLTextures()) 这行代码调用上面讲的子例程载入位图并生成纹理。如果因为任何原因 LoadGLTextures() 调用失败，接着的一行返回FALSE。如果一切OK，并且纹理创建好了，我们启用2D纹理映射。如果您忘记启用的话，您的对象看起来永远都是纯白色，这一定不是什么好事。

```

int InitGL(GLvoid) // 此处开始对OpenGL进行所有设置
{
    if (!LoadGLTextures()) // 调用纹理载入子例程
    {
        return FALSE; // 如果未能载入，返回FALSE
    }

    glEnable(GL_TEXTURE_2D); // 启用纹理映射
    glShadeModel(GL_SMOOTH); // 启用阴影平滑
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // 黑色背景
    glClearDepth(1.0f); // 设置深度缓存
}

```

```

        glEnable(GL_DEPTH_TEST);           // 启用深度测试
        glDepthFunc(GL_LESS);             // 所作深度测试的类型
        glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // 真正精细的透视修正
        return TRUE;                     // 初始化 OK
    }
}

```

现在我们绘制贴图『译者注：其实贴图就是纹理映射。将术语换来换去不好，我想少打俩字。^\_^』过的立方体。这段代码被狂注释了一把，应该很好懂。开始两行代码 glClear() 和 glLoadIdentity() 是第一课中就有的代码。glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT) 清除屏幕并设为我们在 InitGL() 中选定的颜色，本例中是黑色。深度缓存也被清除。模型观察矩阵也使用 glLoadIdentity() 重置。

```

int DrawGLScene(GLvoid)                      // 从这里开始进行所有的绘制
{
    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
    glTranslatef(0.0f,0.0f,-5.0f);              // 移入屏幕 5 个单位
}

```

下面三行使立方体绕X、Y、Z轴旋转。旋转多少依赖于变量 xrot , yrot 和 zrot 的值。

```

glRotatef(xrot,1.0f,0.0f,0.0f);          // 绕X轴旋转
glRotatef(yrot,0.0f,1.0f,0.0f);          // 绕Y轴旋转
glRotatef(zrot,0.0f,0.0f,1.0f);          // 绕Z轴旋转

```

下一行代码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。注意我们在后面是如何使用 glBindTexture 来指定和绑定纹理的。

```
glBindTexture(GL_TEXTURE_2D, texture[0]); // 选择纹理
```

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

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

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

```
glBegin(GL_QUADS);
    // 前面
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // 纹理和四边形的左下
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // 纹理和四边形的右下
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // 纹理和四边形的右上
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // 纹理和四边形的左上
    // 后面
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // 纹理和四边形的右下
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // 纹理和四边形的右上
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // 纹理和四边形的左上
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // 纹理和四边形的左下
    // 顶面
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // 纹理和四边形的左上
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // 纹理和四边形的左下
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // 纹理和四边形的右下
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // 纹理和四边形的右上
    // 底面
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // 纹理和四边形的右上
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // 纹理和四边形的左上
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // 纹理和四边形的左下
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // 纹理和四边形的右下
    // 右面
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // 纹理和四边形的右下
```

```

glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f); // 纹理和四边形的右下
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f); // 纹理和四边形的右上
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f); // 纹理和四边形的左上
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f); // 纹理和四边形的左下
                                // 左面
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f); // 纹理和四边形的左下
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f); // 纹理和四边形的右下
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f); // 纹理和四边形的右上
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f); // 纹理和四边形的左上
glEnd();

```

现在增加 xrot , yrot 和 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。

```

xrot+=0.3f;                                // X 轴旋转
yrot+=0.2f;                                // Y 轴旋转
zrot+=0.4f;                                // Z 轴旋转
return true;                                // 继续运行
}

```

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。！如果您有什么意见或建议请给我EMAIL。如果您认为有什么不对或可以改进，请告诉我。



## 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

## 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

## 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

## 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

## 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。



## 第07课



光照和键盘控制:

在这一课里，我们将添加光照和键盘控制，它让程序看起来更美观。

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。

我们还是在第一课的代码上加以修改。跟以前不一样的是，只要有任何大的改动，我都会写出整段代码。程序开始，我们先加上几个新的变量。

下面几行是新的。我们增加三个布尔变量。light 变量跟踪光照是否打开。变量lp和fp用来存储'L' 和'F'键是否按下的状态。后面我会解释这些变量的重要性。现在，先放在一边吧。

```
BOOL light;           // 光源的开/关  
BOOL lp;             // L键按下了么?  
BOOL fp;             // F键按下了么?
```

现在设置5个变量来控制绕x轴和y轴旋转角度的步长，以及绕x轴和y轴的旋转速度。另外还创建了一个z变量来控制进入屏幕深处的距离。

```
GLfloat xrot;           // X 旋转
GLfloat yrot;           // Y 旋转
GLfloat xspeed;          // X 旋转速度
GLfloat yspeed;          // Y 旋转速度

GLfloat z=-5.0f;         // 深入屏幕的距离
```

接着设置用来创建光源的数组。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alpha通道参数。

因此，下面的代码我们得到的是半亮(0.5f)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。

```
GLfloat LightAmbient[]={ 0.5f, 0.5f, 0.5f, 1.0f };           // 环境光参数
```

下一行代码我们生成最亮的漫射光。所有的参数值都取成最大值1.0f。它将照在我们木板箱的前面，看起来挺好。

```
GLfloat LightDiffuse[]={ 1.0f, 1.0f, 1.0f, 1.0f };           // 漫射光参数
```

最后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0.0f。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0.0f点。所以Z轴上的位移最后定为2.0f。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。『译者注：我很欣赏NeHe的耐心。说真的有时我都打烦了，这么简单的事他这么废话干嘛？但如果什么都清楚，您还会翻着这样的页面看个没完么？』

最后一个参数取为1.0f。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。

```
GLfloat LightPosition[] = { 0.0f, 0.0f, 2.0f, 1.0f }; // 光源位置
```

filter 变量跟踪显示时所采用的纹理类型。第一种纹理(texture 0) 使用gl\_nearest(不光滑)滤波方式构建。第二种纹理 (texture 1) 使用gl\_linear(线性滤波) 方式，离屏幕越近的图像看起来就越光滑。第三种纹理 (texture 2) 使用 mipmaped滤波方式,这将创建一个外观十分优秀的纹理。根据我们的使用类型，filter 变量的值分别等于 0, 1 或 2。下面我们从第一种纹理开始。

GLuint texture[3] 为三种不同纹理分配储存空间。它们分别位于在 texture[0], texture[1] 和 texture[2]中。

```
GLuint filter; // 滤波类型
GLuint texture[3]; // 3种纹理的储存空间
```

现在载入一个位图，并用它创建三种不同的纹理。这一课使用glaux辅助库来载入位图，因此在编译时您应该确认是否包含了glaux库。我知道Delphi和VC++都包含了glaux库，但别的语言不能保证都有。『译者注：glaux是OpenGL辅助库，根据OpenGL的跨平台特性，所有平台上的代码都应通用。但辅助库不是正式的OpenGL标准库，没有出现在所有的平台上。但正好在Win32平台上可用。呵呵，BCB当然也没问题了。』这里我只对新增的代码做注解。如果您对某行代码有疑问的话，请查看教程六。那一课很详细的解释了载入、创建纹理的内容。

在上一段代码后面及ResizeGLScene()之前的位置，我们增加了下面的代码。这和第六课中载入位图的代码几乎相同。

这段代码调用前面的代码载入位图，并将其转换成3个纹理。Status 变量跟踪纹理是否已载入并被创建了。

```
int LoadGLTextures() // 载入位图并转换成纹理
{
    int Status=FALSE; // 状态指示器

    AUX_RGBImageRec *TextureImage[1]; // 创建纹理的存储空间

    memset(TextureImage,0,sizeof(void *)*1); // 将指针设为 NULL
```

现在载入位图并转换成纹理。TextureImage[0]=LoadBMP("Data/Crate.bmp")调用我们的LoadBMP()函数。Data目录下的Crate.bmp将被载入。如果一切正常，图像数据将存放在TextureImage[0]。Status变量被设为TRUE，我们将开始创建纹理。

```
// 载入位图，检查有错，或位图不存在的话退出
if (TextureImage[0]=LoadBMP("Data/Crate.bmp"))
{
    Status=TRUE; // 状态设为 TRUE
```

现在我们已经将图像数据载入TextureImage[0]。我们将用它来创建3个纹理。下面的行告诉OpenGL我们要创建三个纹理，它们将存放在texture[0], texture[1] 和 texture[2] 中。

```
glGenTextures(3, &texture[0]); // 创建纹理
```

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。

```
// 创建 Nearest 滤波贴图
```

```
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, TextureImage[0]->sizeX, TextureImage[0]->sizeY, 0,  
GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
```

The next texture we build is the same type of texture we used in tutorial six. Linear filtered. The only thing that has changed is that we are storing this texture in texture[1] instead of texture[0] because it's our second texture. If we stored it in texture[0] like above, it would overwrite the GL\_NEAREST texture (the first texture).

```
// 创建线性滤波纹理
```

```
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, TextureImage[0]->sizeX, TextureImage[0]->sizeY, 0,  
GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
```

下面是创建纹理的新方法。 Mipmapping!『译者注：这个词的中文我翻不出来，不过没关系。看完这一段，您就知道意思最重要。』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。

```
// 创建 MipMapped 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);
```

下面一行生成 mipmapped 纹理。我们使用三种颜色(红，绿，蓝)来生成一个2D纹理。 TextureImage[0]->sizeX 是位图宽度， textureImage[0]->sizeY 是位图高度， GL\_RGB 意味着我们依次使用RGB色彩。 GL\_UNSIGNED\_BYTE 意味着纹理数据的单位是字节。 TextureImage[0]->data 指向我们创建纹理所用的位图。

```
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, TextureImage[0]->sizeX, TextureImage[0]->sizeY,
GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
}
```

现在释放用来存放位图数据的内存。我们先查看位图数据是否存放在 TextureImage[0] 中，如果有，删掉。然后释放位图结构以确保内存被释放。

```
if (TextureImage[0]) // 纹理是否存在
{
    if (TextureImage[0]->data) // 纹理图像是否存在
    {
        free(TextureImage[0]->data); // 释放纹理图像占用的内存
```

```
    }  
  
    free(TextureImage[0]); // 释放图像结构  
}
```

最后我们返回 status 变量。如果一切OK，status 变量的值为TRUE。否则为FALSE。

```
return Status; // 返回 Status 变量  
}
```

接着应该载入纹理并初始化OpenGL设置了。InitGL函数的第一行使用上面的代码载入纹理。创建纹理之后，我们调用 glEnable(GL\_TEXTURE\_2D) 启用2D纹理映射。阴影模式设为平滑阴影( smooth shading )。背景色设为黑色，我们启用深度测试，然后我们启用优化透视计算。

现在开始设置光源。下面一行设置环境光的发光量，光源light1开始发光。这一课的开始处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)。在int InitGL(GLvoid)函数中添加下面的代码。

```
glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient); // 设置环境光
```

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)。

```
glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse); // 设置漫射光
```

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 - 0.0f，Y - 0.0f，Z方向移向观察者2个单位<位于屏幕外面>)。

```
glLightfv(GL_LIGHT1, GL_POSITION, LightPosition); // 设置光源位置
```

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。

```
glEnable(GL_LIGHT1); // 启用一号光源
```

下一段代码绘制贴图立方体。我只对新增的代码进行注解。如果您对没有注解的代码有  
疑问，回头看看第六课。

```
int DrawGLScene(GLvoid) // 从这里开始进行所有的绘制
{
    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
```

下三行代码放置并旋转贴图立方体。glTranslatef(0.0f,0.0f,z)将立方体沿着Z轴移动z单位。  
glRotatef(xrot,1.0f,0.0f,0.0f)将立方体绕X轴旋转xrot。glRotatef(yrot,0.0f,1.0f,0.0f)将立方体绕  
Y轴旋转yrot。

```
glTranslatef(0.0f,0.0f,z); // 移入/移出屏幕 z 个单位
glRotatef(xrot,1.0f,0.0f,0.0f); // 绕X轴旋转
glRotatef(yrot,0.0f,1.0f,0.0f); // 绕Y轴旋转
```

下一行与我们在第六课中的类似。有所不同的是，这次我们绑定的纹理是texture[filter]，而不是上一课中的texture[0]。任何时候，我们按下F键，filter的值就会增加。如果这个数值大于2，变量filter将被重置为0。程序初始时，变量filter的值也将设为0。使用变量filter我们就可以选择三种纹理中的任意一种。

```
glBindTexture(GL_TEXTURE_2D, texture[filter]);           // 选择由filter决定的纹理
glBegin(GL_QUADS);                                     // 开始绘制四边形
```

glNormal3f是这一课的新东西。Normal就是法线的意思，所谓法线是指经过面(多边形)上的一点且垂直于这个面(多边形)的直线。使用光源的时候必须指定一条法线。法线告诉OpenGL这个多边形的朝向，并指明多边形的正面和背面。如果没有指定法线，什么怪事情都可能发生：不该照亮的面被照亮了，多边形的背面也被照亮.....。对了，法线应该指向多边形的外侧。

看着木箱的前面您会注意到法线与Z轴正向同向。这意味着法线正指向观察者 - 您自己。这正是我们所希望的。对于木箱的背面，也正如我们所要的，法线背对着观察者。如果立方体沿着X或Y轴转个180度的话，前侧面的法线仍然朝着观察者，背面的法线也还是背对着观察者。换句话说，不管是哪个面，只要它朝着观察者这个面的法线就指向观察者。由于光源紧邻观察者，任何时候法线对着观察者时，这个面就会被照亮。并且法线越朝着光源，就显得越亮一些。如果您把观察点放到立方体内部，你就会看到法线里面一片漆黑。因为法线是向外指的。如果立方体内部没有光源的话，当然是一片漆黑。

```
// 前侧面
glNormal3f( 0.0f, 0.0f, 1.0f);                      // 法线指向观察者
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
// 后侧面
glNormal3f( 0.0f, 0.0f, -1.0f);                     // 法线背向观察者
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// 顶面
glNormal3f( 0.0f, 1.0f, 0.0f);                      // 法线向上
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
```

```

glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
// 底面
glNormal3f( 0.0f,-1.0f, 0.0f); // 法线朝下
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
// 右侧面
glNormal3f( 1.0f, 0.0f, 0.0f); // 法线朝右
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
// 左侧面
glNormal3f(-1.0f, 0.0f, 0.0f); // 法线朝左
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd(); // 四边形绘制结束

```

下两行代码将xot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。

```

xrot+=xspeed; // xrot 增加 xspeed 单位
yrot+=yspeed; // yrot 增加 yspeed 单位
return TRUE;
}

```

现在转入WinMain()主函数。我们将在这里增加开关光源、旋转木箱、切换过滤方式以及将木箱移近移远的控制代码。在接近WinMain()函数结束的地方你会看到SwapBuffers(hDC)这行代码。然后就在这一行后面添加如下的代码。

代码将检查L键是否按下过。如果L键已按下，但lp的值不是false的话，意味着L键还没有松开，这时什么都不会发生。

```

SwapBuffers(hDC);           // 交换缓存
if (keys['L'] && !lp)      // L 键已按下并且松开了?
{

```

如果lp的值是false的话，意味着L键还没按下，或者已经松开了，接着lp将被设为TRUE。同时检查这两个条件的原因是为了防止L键被按住后，这段代码被反复执行，并导致窗体不停闪烁。

lp设为true之后，计算机就知道L键按过了，我们则据此可以切换光源的开/关：布尔变量light控制光源的开关。

```

lp=TRUE;                   // lp 设为 TRUE
light=!light;              // 切换光源的 TRUE/FALSE

```

Now we check to see what light ended up being. The first line translated to english means: If light equals false. So if you put it all together, the lines do the following: If light equals false, disable lighting. This turns all lighting off. The command 'else' translates to: if it wasn't false. So if light wasn't false, it must have been true, so we turn lighting on.

```

if (!light)                // 如果没有光源
{
    glDisable(GL_LIGHTING); // 禁用光源
}
else                      // 否则
{
    glEnable(GL_LIGHTING); // 启用光源
}

```

下面的代码查看是否松开了"L"键。如果松开，变量lp将设为false。这意味着"L"键没有按下。如果不作此检查，光源第一次打开之后，就无法再关掉了。计算机会以为"L"键一直按着呢。

```

if (!keys['L'])
{
    fp=FALSE;           // L键松开了么?
}

```

然后对"F"键作相似的检查。如果有按下"F"键并且"F"键没有处于按着的状态或者它就从没有按下过，将变量fp设为true。这意味着这个键正被按着呢。接着将filter变量加一。如果filter变量大于2(因为这里我们的使用的数组是texture[3],大于2的纹理不存在)，我们重置filter变量为0。

```

if (keys['F'] && !fp)           // F键按下了么?
{
    fp=TRUE;                   // fp 设为 TRUE
    filter+=1;                 // filter的值加一
    if (filter>2)
    {
        filter=0;             // 若是重置为0
    }
}
if (!keys['F'])                 // F键放开了么?
{
    fp=FALSE;                  // 若是fp设为FALSE
}

```

这四行检查是否按下了PageUp键。若是的话，减少z变量的值。这样DrawGLScene函数中包含的glTranslatef(0.0f,0.0f,z)调用将使木箱离观察者更远一点。

```

if (keys[VK_PRIOR])           // PageUp按下了?
{
    z-=0.02f;                  // 若按下，将木箱移向屏幕内部
}

```

接着四行检查PageDown键是否按下，若是的话，增加z变量的值。这样DrawGLScene函数中包含的glTranslatef(0.0f,0.0f,z)调用将使木箱向着观察者移近一点。

```

if (keys[VK_NEXT])           // PageDown按下了么
{
    z+=0.02f;               // 若按下的话，将木箱移向观察者
}

```

现在检查方向键。按下左右方向键xspeed相应减少或增加。按下上下方向键yspeed相应减少或增加。记住在以后的教程中如果xspeed、yspeed的值增加的话，立方体就转的更快。如果一直按着某个方向键，立方体会在那个方向上转的越快。

```

if (keys[VK_UP])            // Up方向键按下了么?
{
    xspeed-=0.01f;          // 若是,减少xspeed
}
if (keys[VK_DOWN])          // Down方向键按下了么?
{
    xspeed+=0.01f;          // 若是,增加xspeed
}
if (keys[VK_RIGHT])         // Right方向键按下了么?
{
    yspeed+=0.01f;          // 若是,增加yspeed
}
if (keys[VK_LEFT])          // Left方向键按下了么?
{
    yspeed-=0.01f;          // 若是,减少yspeed
}

```

像前几课一样，我们最后还需要更正窗体的标题。

```

if (keys[VK_F1])            // F1按下了么?
{
    keys[VK_F1]=FALSE;       // 若是将其设为FALSE
    KillGLWindow();          // 销毁当前窗口
}

```

```
fullscreen=!fullscreen;           // 切换全屏/窗口模式
// 重建GL窗口
if (!CreateGLWindow("NeHe's Textures, Lighting & Keyboard Tutorial",640,480,16,
fullscreen))
{
    return 0;                  // 若无法创建窗口，程序退出
}
}

}

// 关闭
KillGLWindow();                // 销毁窗口
return (msg.wParam);           // 退出程序
}
```

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。

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

#### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我



都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

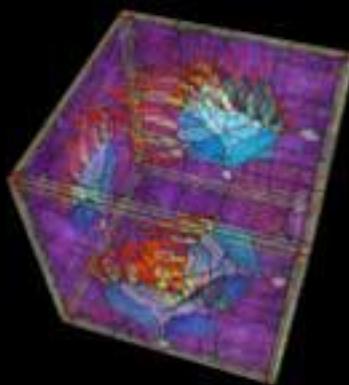
感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第06课

第08课 &gt;



## 第08课



混合:

在这一课里，我们在纹理的基础上加上了混合，它看起具有透明的效果，当然解释它不是那么容易，当希望你喜欢它。

### 简单的透明

OpenGL中的绝大多数特效都与某些类型的(色彩)混合有关。混色的定义为，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/或者所使用的混色函数。Alpha通常是位于颜色值末尾的第4个颜色组分量。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。

### 混色的公式

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)混合的工作原理，这一节应该适合您吧。『译者注:其实并不难^-^。原文中的公式如下，CKER再唠叨一下吧。其实混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 - 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混合之后，最后将混合得到的RGB分量重新合并。』

公式如下：

( $R_s S_r + R_d D_r, G_s S_g + G_d D_g, B_s S_b + B_d D_b, A_s A_s + A_d D_a$ )

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

$(R_s A_s + R_d (1 - A_s), G_s A_s + G_d (1 - A_s), B_s A_s + B_d (1 - A_s), A_s A_s + A_d (1 - A_s))$

这个公式会生成透明/半透明的效果。

## OpenGL中的混色

在OpenGL中实现混色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：正确的混色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(先画最近的物体)。

考虑对两个多边形(1和2)进行alpha混合，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。

我们将使用第七课的代码。一开始先在代码开始处增加两个新的变量。出于清晰起见，我重写了整段代码。

```
bool blend;           // 是否混合?
bool bp;              // B 键按下了么?
```

然后往下移动到 LoadGLTextures() 这里。找到" if (TextureImage[0]=LoadBMP("Data/Crate.bmp")) "这一行。我们现在使用有色玻璃纹理来代替上一课中的木箱纹理。

```
if (TextureImage[0]=LoadBMP("Data/glass.bmp")) // 载入玻璃位图
```

在InitGL()代码段加入以下两行。第一行以全亮度绘制此物体，并对其进行50%的alpha混合(半透明)。当混合选项打开时，此物体将会产生50%的透明效果。第二行设置所采用的混合类型。

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。

```
glColor4f(1.0f,1.0f,1.0f,0.5f);           // 全亮度， 50% Alpha 混合
glBlendFunc(GL_SRC_ALPHA,GL_ONE);          // 基于源象素alpha通道值的半透明混合函数
```

在接近第七课结尾处的地方找到下面的代码段。

```
if (keys[VK_LEFT])                  // Left方向键按下了么?
{
    yspeed-=0.01f;                 // 若是, 减少yspeed
}
```

接着上面的代码，我们增加如下的代码。这几行监视B键是否按下。如果是的话，计算机检查混合选项是否已经打开。然后将其置为相反的状态。

```
if (keys['B'] && !bp)            // B 键按下且bp为 FALSE么?
{
    bp=TRUE;                      // 若是， bp 设为 TRUE
    blend = !blend;                // 切换混合选项的 TRUE / FALSE
    if(blend)                     // 混合打开了么?
    {
        glEnable(GL_BLEND);       // 打开混合
        glDisable(GL_DEPTH_TEST); // 关闭深度测试
    }
    else                          // 否则
    {
        glDisable(GL_BLEND);     // 关闭混合
        glEnable(GL_DEPTH_TEST); // 打开深度测试
    }
}
```

```

}
if (!keys['B'])           // B 键松开了么?
{
    bp=FALSE;            // 若是 , bp设为 FALSE
}

```

但是怎样才能在使用纹理贴图的时候指定混合时的颜色呢?很简单，在调整贴图模式时，纹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混合的确很简单！

### 原文注 (11/13/99)

我(NeHe)混色代码进行了修改，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参数来混合，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。我所用的混色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混合

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。

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

#### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。



### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

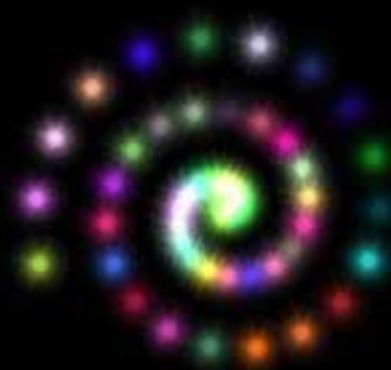
感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第07课

第09课 &gt;



## 第09课



### 3D空间中移动图像：

你想知道如何在3D空间中移动物体，你想知道如何在屏幕上绘制一个图像，而让图像的背景色变为透明，你希望有一个简单的动画。这一课将教会你所有的一切。前面的课程涵盖了基础的OpenGL，每一课都是在前一课的基础上创建的。这一课是前面几课知识的综合，当你学习这课时，请确保你已经掌握了前面几课的知识。

欢迎进入第九课。到现在为止，您应该很好的理解OpenGL了。『CKER：如果没有的话，一定是我翻译的罪过……』。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混色(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3D场景中移动位图，并去除位图上的黑色象素(使用混色)。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混合，得到简单的动画效果。

我们在第一课的代码基础上进行修改。先在程序源码的开始处增加几个变量。出于清晰起见，我重写了整段代码。

```
#include    <stdio.h>           // 标准输入输出库头文件  
#include    <glaux.h>          // GLaux库的头文件
```

下列这几行新加的。twinkle和tp是布尔变量, 表示它们只能设为 TRUE 或 FALSE。twinkle用来跟踪闪烁效果是否启用。tp用来检查 'T' 键有没有被按下或松开. (按下时 tp=TRUE, 松开时 tp=FALSE)。

```
BOOL twinkle; // 闪烁的星星
BOOL tp; // 'T' 按下了么?
```

num 跟踪屏幕上所绘制的星星数。这个数字被定义为一个常量。这意味着无法在以后的代码中对其进行修改。这么做的原因是您无法重新定义一个数组。因此，如果我们定义一个50颗星星的数组，然后又将num增加到51的话，就会出错『CKER：数组越界』。不过您还是可以(也只可以)在这一行上随意修改这个数字。但是以后请您别再改动 num 的值了，除非您想看见灾难发生。

```
const num=50; // 绘制的星星数
```

现在我们来创建一个结构。结构这词听起来有点可怕，但实际上并非如此。一个结构使用一组简单类型的数据(以及变量等)来表达较大的具有相似性的数据组合。我们知道我们在保持对星星的跟踪。您可以看到下面的第七行就是 stars；并且每个星星有三个整型的色彩值。第三行 int r,g,b设置了三个整数. 一个红色(r), 一个绿色(g), 以及一个蓝色(b). 此外，每个星星离屏幕中心的距离不同，而且可以是以屏幕中心为原点的任意360度中的一个角度。如果你看下面第四行的话，会发现我们使用了一个叫做 dist的浮点数来保持对距离的跟踪. 第五行则用一个叫做 angle的浮点数保持对星星角度值的跟踪。因此我们使用了一组数据来描述屏幕上星星的色彩, 距离, 和角度。不幸的是我们不止对一个星星进行跟踪。但是无需创建 50 个红色值、50 个绿色值、50 个蓝色值、50 个距离值以及 50 个角度值，而只需创建一个数组star。star数组的每个元素都是stars类型的，里面存放了描述星星的所有数据。star数组在下面的第八行创建。第八行的样子是这样的：stars star[num]。数组类型是 stars结构. 所数组能存放所有stars结构的信息。数组名字是 star. 数组大小是 [num]。数组中存放着 stars结构的元素. 跟踪结构元素会比跟踪各自分开的变量容易的多. 不过这样也很笨, 因为我们竟然不能改变常量 num 来增减星星数量。

```
typedef struct // 为星星创建一个结构
{
    int r, g, b; // 星星的颜色
```

```

GLfloat dist;           // 星星距离中心的距离
GLfloat angle;          // 当前星星所处的角度
}
stars;                  // 结构命名为stars
stars star[num];       // 使用 'stars' 结构生成一个包含 'num' 个元素的 'star' 数组

```

接下来我们设置几个跟踪变量：星星离观察者的距离变量(zoom)，我们所见到的星星所处的角度(tilt)，以及使闪烁的星星绕Z轴自转的变量spin。

loop变量用来绘制50颗星星。texture[1]用来存放一个黑白纹理。如果您需要更多的纹理的话，您应该增加texture数组的大小至您决定采用的纹理个数。

```

GLfloat zoom=-15.0f;           // 星星离观察者的距离
GLfloat tilt=90.0f;            // 星星的倾角
GLfloat spin;                 // 闪烁星星的自转

GLuint loop;                  // 全局 Loop 变量
GLuint texture[1];           // 存放一个纹理

```

紧接着上面的代码就是我们用来载入纹理的代码。我不打算再详细的解释这段代码。这跟我们在第六、七、八课中所用的代码是一模一样的。这一次载入的位图叫做star.bmp。这里我们使用glGenTextures(1, &texture[0])，来生成一个纹理。纹理采用线性滤波方式。

```

AUX_RGBImageRec *LoadBMP(char *Filename)           // 载入位图文件
{
    FILE *File=NULL;                            // 文件句柄
    if (!Filename)                            // 确认已给出文件名
    {
        return NULL;                          // 若无返回 NULL
    }

    File=fopen(Filename,"r");                // 检查文件是否存在
    if (File)                                // 文件存在么?
    {

```

```

    fclose(File);           // 关闭文件句柄
    return auxDIBImageLoad(Filename); // 载入位图并返回指针
}
return NULL;           // 如果载入失败返回 NULL
}

```

下面的代码(调用上面的代码)载入位图，并转换成纹理。变量用来跟踪纹理是否已载入并创建好了。

```

int LoadGLTextures()           // 载入位图并转换成纹理
{
    int Status=FALSE;          // 状态指示器
    AUX_RGBImageRec *TextureImage[1]; // 为纹理分配存储空间
    memset(TextureImage,0,sizeof(void *)*1); // 将指针设为 NULL

    // 载入位图，查错，如果未找到位图文件则退出
    if (TextureImage[0]=LoadBMP("Data/Star.bmp"))
    {
        Status=TRUE;           // 将 Status 设为 TRUE
        glGenTextures(1, &texture[0]); // 创建一个纹理

        // 创建一个线性滤波纹理
        glBindTexture(GL_TEXTURE_2D, texture[0]);
        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, TextureImage[0]->sizeX, TextureImage[0]->sizeY, 0,
        GL_RGB, GL_UNSIGNED_BYTE, TextureImage[0]->data);
    }

    if (TextureImage[0])           // 如果纹理存在
    {
        if (TextureImage[0]->data) // 如果纹理图像存在
        {
            free(TextureImage[0]->data); // 释放纹理图像所占的内存
        }
    }

    free(TextureImage[0]);         // 释放图像结构
}

```

```

        }

    return Status;           // 返回 Status的值
}

```

现在设置OpenGL的渲染方式。这里不打算使用深度测试，如果您使用第一课的代码的话，请确认是否已经去掉了glDepthFunc(GL\_LESS); 和 glEnable(GL\_DEPTH\_TEST); 两行。否则，您所见到的效果将会一团糟。这里我们使用了纹理映射，因此请您确认您已经加上了这些第一课中所没有的代码。您会注意到我们通过混色来启用了纹理映射。

```

int InitGL(GLvoid)          // 此处开始对OpenGL进行所有设置
{
    if (!LoadGLTextures())   // 调用纹理载入子例程
    {
        return FALSE;        // 如果未能载入，返回FALSE
    }

    glEnable(GL_TEXTURE_2D); // 启用纹理映射
    glShadeModel(GL_SMOOTH); // 启用阴影平滑
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f); // 黑色背景
    glClearDepth(1.0f);      // 设置深度缓存
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // 真正精细的透视修正
    glBlendFunc(GL_SRC_ALPHA,GL_ONE); // 设置混色函数取得半透明效果
    glEnable(GL_BLEND);       // 启用混色
}

```

以下是新增的代码。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]的角度我们所做的只是star[1].angle={某个数值}；就这么简单！

```

for (loop=0; loop<num; loop++)          // 创建循环设置全部星星
{
    star[loop].angle=0.0f;                // 所有星星都从零角度开始
}

```

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5.0f。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1.0f。之所以要乘以5.0f是因为 $1.0f * 5.0f$ 就是5.0f。『CKER：废话，废话！这老外怎么跟孔乙己似的！:)』5.0f已经很接近屏幕边缘。我不想星星飞出屏幕，5.0f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5.0f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0.0f~1.0f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。

```

star[loop].dist=(float(loop)/num)*5.0f;    // 计算星星离中心的距离
star[loop].r=rand()%256;                    // 为star[loop]设置随机红色分量
star[loop].g=rand()%256;                    // 为star[loop]设置随机绿色分量
star[loop].b=rand()%256;                    // 为star[loop]设置随机蓝色分量
}
return TRUE;                                // 初始化一切OK
}

```

Resize的代码也是一样的，现在我们转入绘图代码。如果您使用第一课的代码，删除旧的DrawGLScene代码，只需将下面的代码复制过去就行了。实际上，第一课的代码只有两行，所以没太多东西要删掉的。

```

int DrawGLScene(GLvoid)                      // 此过程中包括所有的绘制代码
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除屏幕及深度缓存
    glBindTexture(GL_TEXTURE_2D, texture[0]);      // 选择纹理

    for (loop=0; loop<num; loop++)                // 循环设置所有的星星
    {
        glLoadIdentity();                          // 绘制每颗星星之前，重置模型观察矩阵
        glTranslatef(0.0f,0.0f,zoom);            // 深入屏幕里面
        glRotatef(tilt,1.0f,0.0f,0.0f);          // 倾斜视角
    }
}

```

现在我们来移动星星。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。

第二行代码沿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,0.0f,1.0f,0.0f); // 旋转至当前所画星星的角度  
glTranslatef(star[loop].dist,0.0f,0.0f); // 沿X轴正向移动
```

接着的代码带点小技巧。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。下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。

```
glRotatef(-star[loop].angle,0.0f,1.0f,0.0f); // 取消当前星星的角度  
glRotatef(-tilt,1.0f,0.0f,0.0f); // 取消屏幕倾斜
```

如果 twinkle 为 TRUE , 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。(如果您这里看不懂的话 , 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。

```
if (twinkle)          // 启用闪烁效果
{
    // 使用byte型数值指定一个颜色
    glColor4ub(star[(num-loop)-1].r,star[(num-loop)-1].g,star[(num-loop)-1].b,255);
    glBegin(GL_QUADS);           // 开始绘制纹理映射过的四边形
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 0.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 0.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 0.0f);
    glEnd();                  // 四边形绘制结束
}
```

现在绘制第二遍的星星。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。

```
glRotatef(spin,0.0f,0.0f,1.0f);      // 绕z轴旋转星星
// 使用byte型数值指定一个颜色
	glColor4ub(star[loop].r,star[loop].g,star[loop].b,255);
	glBegin(GL_QUADS);           // 开始绘制纹理映射过的四边形
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 0.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 0.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 0.0f);
    glEnd();                  // 四边形绘制结束
```

以下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。

```
spin+=0.01f;           // 星星的公转
star[loop].angle+=float(loop)/num; // 改变星星的自转角度
star[loop].dist-=0.01f; // 改变星星离中心的距离
```

接着几行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。

```
if (star[loop].dist<0.0f)           // 星星到达中心了么
{
    star[loop].dist+=5.0f;          // 往外移5个单位
    star[loop].r=rand()%256;        // 赋一个新红色分量
    star[loop].g=rand()%256;        // 赋一个新绿色分量
    star[loop].b=rand()%256;        // 赋一个新蓝色分量
}
return TRUE;                      // 一切正常
}
```

现在我们添加监视键盘的代码。下移到WinMain()。找到SwapBuffers(hDC)一行。我们就在这一行后面增加键盘监视代码。

代码将检查T键是否已按下。如果T键按下过，并且又放开了，if块内的代码将被执行。如果twinkle为FALSE，他将变为TRUE。反之亦然。只要T键按下，tp就变为TRUE。这样处理可以防止如果您一直按着T键的话，块内的代码被反复执行。

```
SwapBuffers(hDC);           // 切换缓冲
if (keys['T'] && !tp)      // 是否T键已按下并且 tp值为 FALSE
{
    tp=TRUE;                // 若是，将tp设为TRUE
```

```
twinkle=!twinkle;           // 翻转 twinkle的值  
}
```

下面的代码检查是否松开了T键。若是，使 tp=FALSE。除非tp的值为FALSE，否则按着T键时什么也不会发生。所以这行代码很重要。

```
if (!keys['T'])           // T 键已松开了么?  
{  
    tp=FALSE;             // 若是，tp为 FALSE  
}
```

余下的代码检查上、下方向键，向上翻页键或向下翻页键是否按下。

```
if (keys[VK_UP])          // 上方向键按下了么?  
{  
    tilt-=0.5f;            // 屏幕向上倾斜  
}  
  
if (keys[VK_DOWN])         // 下方向键按下了么?  
{  
    tilt+=0.5f;            // 屏幕向下倾斜  
}  
  
if (keys[VK_PRIOR])        // 向上翻页键按下了么  
{  
    zoom-=0.2f;             // 缩小  
}  
  
if (keys[VK_NEXT])          // 向下翻页键按下了么?  
{  
    zoom+=0.2f;             // 放大  
}
```

像以前一样，确认窗口的标题是否正确。

```
if (keys[VK_F1])           // F1键按下了么?  
{  
    keys[VK_F1]=FALSE;     // 若是，使对应的Key数组中的值为 FALSE  
    KillGLWindow();        // 销毁当前的窗口  
    fullscreen=!fullscreen; // 切换 全屏 / 窗口 模式  
    // 重建 OpenGL 窗口  
    if (!CreateGLWindow("NeHe's 透明纹理实例",640,480,16,fullscreen))  
    {  
        return 0;           // 如果窗口未能创建，程序退出  
    }  
}  
}  
}
```

这一课我尽我所能来解释如何加载一个灰阶位图纹理，(使用混色)去掉它的背景色后，再给它上色，最后让它在3D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！

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

#### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明



所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

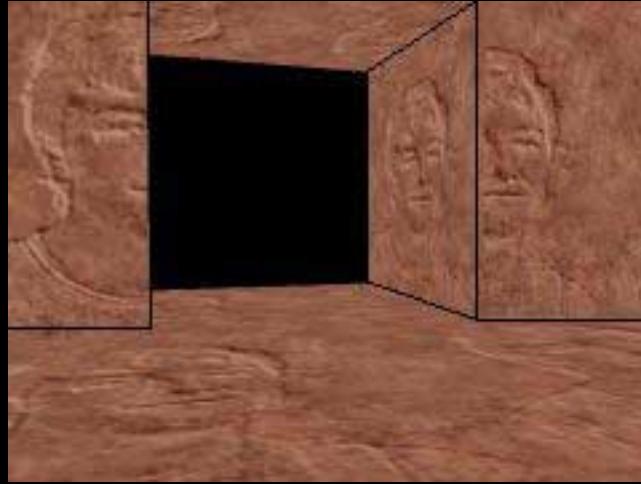
感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第08课

第10课 &gt;



## 第10课



加载3D世界，并在其中漫游：

在这一课中，你将学会如何加载3D世界，并在3D世界中漫游。这一课使用第一课的代码，当然在课程说明中我只介绍改变了代码。

这一课是由Lionel Brits ( telgeuse)所写的。在本课中我们只对增加的代码做解释。当然只添加课程中所写的代码，程序是不会运行的。如果您有兴趣知道下面的每一行代码是如何运行的话，请下载完整的源码，并在浏览这一课的同时，对源码进行跟踪。

好了现在欢迎来到名不见经传的第十课。到现在为止，您应该有能力创建一个旋转的立方体或一群星星了，对3D编程也应该有些感觉了吧？但还是请等一下！不要立马冲动地要开始写个Quake

IV，好不好....)。只靠旋转的立方体还很难来创造一个可以决一死战的酷毙了的对手....)。现在这些日子您所需要的是一个大一点的、更复杂些的、动态3D世界，它带有空间的六自由度和花哨的效果如镜像、入口、扭曲等等，当然还要有更快的帧显示速度。这一课就要解释一个基本的3D世界"结构"，以及如何在这个世界里游走。

### 数据结构

当您想要使用一系列的数字来完美的表达3D环境时，随着环境复杂度的上升，这个工作的难度也会随之上升。出于这个原因，我们必须将数据归类，使其具有更多的可操作性风格。在程序清单头部出现了sector(区段)的定义。每个3D世界基本上可以看作是sector(区段)的集合。一个sector(区段)可以是一个房间、一个立方体、或者任意一个闭合的区间。

```
typedef struct tagSECTOR // 创建Sector区段结构
{
    int numtriangles; // Sector中的三角形个数
    TRIANGLE* triangle; // 指向三角数组的指针
} SECTOR; // 命名为SECTOR
```

一个sector(区段)包含了一系列的多边形，所以下一个目标就是triangle(我们将只用三角形，这样写代码更容易些)。

```
typedef struct tagTRIANGLE // 创建Triangle三角形结构
{
    VERTEX vertex[3]; // VERTEX矢量数组，大小为3
} TRIANGLE; // 命名为 TRIANGLE
```

三角形本质上是由一些(两个以上)顶点组成的多边形，顶点同时也是我们的最基本的分类单位。顶点包含了OpenGL真正感兴趣的数据。我们用3D空间中的坐标值(x,y,z)以及它们的纹理坐标(u,v)来定义三角形的每个顶点。

```
typedef struct tagVERTEX // 创建Vertex顶点结构
{
    float x, y, z; // 3D 坐标
    float u, v; // 纹理坐标
} VERTEX; // 命名为VERTEX
```

## 载入文件

在程序内部直接存储数据会让程序显得太过死板和无趣。从磁盘上载入世界资料，会给我们带来更多的弹性，可以让我们体验不同的世界，而不用被迫重新编译程序。另一个好处就是用户可以切换世界资料并修改它们而无需知道程序如何读入输出这些资料的。数据文件的类型我们准备使用文本格式。这样编辑起来更容易，写的代码也更少。等将来我们也许会使用二进制文件。

问题是，怎样才能从文件中取得数据资料呢？首先，创建一个叫做SetupWorld()的新函数。把这个文件定义为filein，并且使用只读方式打开文件。我们必须在使用完毕之后关闭文件。大家一起来看看现在的代码：

```
// 先前的定义：char* worldfile = "data\\world.txt";
void SetupWorld() // 设置我们的世界
{
    FILE *filein; // 工作文件
    filein = fopen(worldfile, "rt"); // 打开文件

    ...
    (读入数据资料))
    ...

    fclose(filein); // 关闭文件
    return; // 返回
}
```

下一个挑战是将每个单独的文本行读入变量。这有很多办法可以做到。一个问题是文件中并不是所有的行都包含有意义的信息。空行和注释不应该被读入。我们创建了一个叫做readstr()的函数。这个函数会从数据文件中读入一个有意义的行至一个已经初始化过的字符串。下面就是代码：

```
void readstr(FILE *f, char *string) // 读入一个字符串
{
    do // 循环开始
    {
        fgets(string, 255, f); // 读入一行
    } while ((string[0] == '/') || (string[0] == '\n')); // 考察是否有必要进行处理
    return; // 返回
}
```

}

下一步我们读入区段数据。这一课将只处理一个区段，不过实现一个多区段引擎也很容易。让我们将注意力转回SetupWorld()。程序必须知道区段内包含了多少个三角形。我们在数据文件中以下面这种形式定义三角形数量：

接下来是读取三角形数量的代码：

```
int numtriangles; // 区段中的三角形数量
char oneline[255]; // 存储数据的字符串
...
readstr(filein,oneline); // 读入一行数据
sscanf(oneline, "NUMPOLIES %d\n", &numtriangles); // 读入三角形数量
```

余下的世界载入过程采用了相似的方法。接着，我们对区段进行初始化，并读入部分数据：

```
// 先前的定义: SECTOR sector1;
char oneline[255]; // 存储数据的字符串
int numtriangles; // 区段的三角形数量
float x, y, z, u, v; // 3D 和 纹理坐标
...
sector1.triangle = new TRIANGLE[numtriangles]; // 为numtriangles个三角形分配内存并设定指针
sector1.numtriangles = numtriangles; // 定义区段1中的三角形数量
// 遍历区段中的每个三角形
for (int triloop = 0; triloop < numtriangles; triloop++) // 遍历所有的三角形
{
    // 遍历三角形的每个顶点
    for (int vertloop = 0; vertloop < 3; vertloop++) // 遍历所有的顶点
    {
        readstr(filein,oneline); // 读入一行数据
        // 读入各自的顶点数据
        sscanf(oneline, "%f %f %f %f", &x, &y, &z, &u);
        // 将顶点数据存入各自的顶点
        sector1.triangle[triloop].vertex[vertloop].x = x; // 区段 1, 第 triloop 个三角形, 第 vertloop 个顶点, 值 x=x
```

```

    sector1.triangle[triloop].vertex[vertloop].y = y; // 区段 1, 第 triloop 个三角形, 第 vertloop 个
顶点, 值 y =y
    sector1.triangle[triloop].vertex[vertloop].z = z; // 区段 1, 第 triloop 个三角形, 第 vertloop 个
顶点, 值 z =z
    sector1.triangle[triloop].vertex[vertloop].u = u; // 区段 1, 第 triloop 个三角形, 第 vertloop 个
顶点, 值 u =u
    sector1.triangle[triloop].vertex[vertloop].v = v; // 区段 1, 第 triloop 个三角形, 第 vertloop 个
顶点, 值 e=v
}
}

```

数据文件中每个三角形都以如下形式声明:

```

X1 Y1 Z1 U1 V1
X2 Y2 Z2 U2 V2
X3 Y3 Z3 U3 V3

```

### 显示世界

现在区段已经载入内存，我们下一步要在屏幕上显示它。到目前为止，我们所作过的都是些简单的旋转和平移。但我们的镜头始终位于原点(0, 0, 0)处。任何一个不错的3D引擎都会允许用户在这个世界中游走和遍历，我们的这个也一样。实现这个功能的一种途径是直接移动镜头并绘制以镜头为中心的3D环境。这样做会很慢并且不易用代码实现。我们的解决方法如下：

根据用户的指令旋转并变换镜头位置。

围绕原点，以与镜头相反的旋转方向来旋转世界。(让人产生镜头旋转的错觉)  
以与镜头平移方式相反的方式来平移世界(让人产生镜头移动的错觉)。

这样实现起来就很简单。

下面从第一步开始吧(平移并旋转镜头)。

```

if (keys[VK_RIGHT]) // 右方向键按下了么?
{
    yrot -= 1.5f; // 向左旋转场景
}

if (keys[VK_LEFT]) // 左方向键按下了么?
{
    yrot += 1.5f; // 向右侧旋转场景
}

```

```

if (keys[VK_UP]) // 向上方向键按下了么?
{
    xpos -= (float)sin(heading*piover180) * 0.05f; // 沿游戏者所在的X平面移动
    zpos -= (float)cos(heading*piover180) * 0.05f; // 沿游戏者所在的Z平面移动
    if (walkbiasangle >= 359.0f) // 如果walkbiasangle大于359度
    {
        walkbiasangle = 0.0f; // 将 walkbiasangle 设为0
    }
    else // 否则
    {
        walkbiasangle += 10; // 如果 walkbiasangle < 359 , 则增加 10
    }
    walkbias = (float)sin(walkbiasangle * piover180)/20.0f; // 使游戏者产生跳跃感
}

if (keys[VK_DOWN]) // 向下方向键按下了么 ?
{
    xpos += (float)sin(heading*piover180) * 0.05f; // 沿游戏者所在的X平面移动
    zpos += (float)cos(heading*piover180) * 0.05f; // 沿游戏者所在的Z平面移动
    if (walkbiasangle <= 1.0f) // 如果walkbiasangle小于1度
    {
        walkbiasangle = 359.0f; // 使 walkbiasangle 等于 359
    }
    else // 否则
    {
        walkbiasangle -= 10; // 如果 walkbiasangle > 1 减去 10
    }
    walkbias = (float)sin(walkbiasangle * piover180)/20.0f; // 使游戏者产生跳跃感
}

```

这个实现很简单。当左右方向键按下后，旋转变量yrot相应增加或减少。当前后方向键按下后，我们使用sine和cosine函数重新生成镜头位置(您需要些许三角函数学的知识:-)。Piover180是一个很简单的折算因子用来折算度和弧度。

接着您可能会问：walkbias是什么意思？这是NeHe的发明的单词:-)。基本上就是当人行走时头部产生上下摆动的幅度。我们使用简单的sine正弦波来调节镜头的Y轴位置。如果不添加这个而只是前后移动的话，程序看起来就没这么棒了。

现在，我们已经有了下面这些变量。可以开始进行步骤2和3了。由于我们的程序还不太复杂，我们无需新建一个函数，而是直接在显示循环中完成这些步骤。

```

int DrawGLScene(GLvoid) // 绘制 OpenGL 场景
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除 场景 和 深度缓冲
    glLoadIdentity(); // 重置当前矩阵

    GLfloat x_m, y_m, z_m, u_m, v_m; // 顶点的临时 X, Y, Z, U 和 V 的数值
    GLfloat xtrans = -xpos; // 用于游戏者沿X轴平移时的大小
    GLfloat ztrans = -zpos; // 用于游戏者沿Z轴平移时的大小
    GLfloat ytrans = -walkbias-0.25f; // 用于头部的上下摆动
    GLfloat sceneroty = 360.0f - yrot; // 位于游戏者方向的360度角

    int numtriangles; // 保有三角形数量的整数

    glRotatef(lookupdown,1.0f,0,0); // 上下旋转
    glRotatef(sceneroty,0,1.0f,0); // 根据游戏者正面所对方向所作的旋转

    glTranslatef(xtrans, ytrans, ztrans); // 以游戏者为中心的平移场景
    glBindTexture(GL_TEXTURE_2D, texture[filter]); // 根据 filter 选择的纹理

    numtriangles = sector1.numtriangles; // 取得Sector1的三角形数量

    // 逐个处理三角形
    for (int loop_m = 0; loop_m < numtriangles; loop_m++) // 遍历所有的三角形
    {
        glBegin(GL_TRIANGLES); // 开始绘制三角形
        glNormal3f( 0.0f, 0.0f, 1.0f); // 指向前面的法线
        x_m = sector1.triangle[loop_m].vertex[0].x; // 第一点的 X 分量
        y_m = sector1.triangle[loop_m].vertex[0].y; // 第一点的 Y 分量
        z_m = sector1.triangle[loop_m].vertex[0].z; // 第一点的 Z 分量
        u_m = sector1.triangle[loop_m].vertex[0].u; // 第一点的 U 纹理坐标
        v_m = sector1.triangle[loop_m].vertex[0].v; // 第一点的 V 纹理坐标
        glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m); // 设置纹理坐标和顶点

        x_m = sector1.triangle[loop_m].vertex[1].x; // 第二点的 X 分量
        y_m = sector1.triangle[loop_m].vertex[1].y; // 第二点的 Y 分量
        z_m = sector1.triangle[loop_m].vertex[1].z; // 第二点的 Z 分量
        u_m = sector1.triangle[loop_m].vertex[1].u; // 第二点的 U 纹理坐标
        v_m = sector1.triangle[loop_m].vertex[1].v; // 第二点的 V 纹理坐标
        glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m); // 设置纹理坐标和顶点
    }
}

```

```

x_m = sector1.triangle[loop_m].vertex[2].x; // 第三点的 X 分量
y_m = sector1.triangle[loop_m].vertex[2].y; // 第三点的 Y 分量
z_m = sector1.triangle[loop_m].vertex[2].z; // 第三点的 Z 分量
u_m = sector1.triangle[loop_m].vertex[2].u; // 第二点的 U 纹理坐标
v_m = sector1.triangle[loop_m].vertex[2].v; // 第二点的 V 纹理坐标
glTexCoord2f(u_m,v_m); glVertex3f(x_m,y_m,z_m); // 设置纹理坐标和顶点
glEnd(); // 三角形绘制结束
}
return TRUE; // 返回
}

```

搞定！我们已经完成了自己的第一帧画面。这绝对算不上什么Quake，但咳...，我们绝对也不是Carmack或者Abrash。运行程序时，您可以按下F、B、

PgUp 和 PgDown 键来看看效果。PgUp /

PgDown简单的上下倾斜镜头。如果NeHe决定保留的话，程序中使用的纹理取自于我的学校ID证件上的照片，并且做了浮雕效果.....)。

现在您也许在考虑下一步该做什么。但还是不要考虑使用这些代码来实现完整的3D引擎，写这个程序的目的也并非如此。您也许希望您的游戏中不止存在一个Sector，尤其是实现类似入口这样的部分，您还可能需要使用多边形(超过3个顶点)。程序现在的代码实现允许载入多个Sector并剔除了背面(背向镜头不用绘制的多边形)。将来我会写个这样的教程，但这需要更多的数学知识基础。

### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自



己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第09课

第11课 &gt;



## 第11课



飘动的旗帜:

这一课从第六课的代码开始，创建一个飘动的旗帜。我相信在这课结束的时候，你可以掌握纹理映射和混合操作。

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演示（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。

在我们开始之前，先打开Visual C++（译者：我可是用的C++ Builder...）并在其他的#include之后，添加如下的代码。这将引入我们在程序中将要用到的复杂（译者：复杂吗？）数学函数sin和cosine。

```
#include <math.h> // 引入数学函数库中的Sin
```

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 $45 \times 45$ 点形成，换句话说也就是由 $44$ 格 $\times 44$ 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。这几行添加在程序头部，位于最后一行#include之后、GLuint texture[1]之前的位置。

```
float points[ 45 ][ 45 ][3];           // Points网格顶点数组
int wiggle_count = 0;                   // 指定旗形波浪的运动速度
GLfloat hold;                         // 临时变量
```

然后下移至LoadGLTextures()子过程。本课中使用的纹理文件名是Tim.bmp。找到LoadBMP("Data/NeHe.bmp")这一句，并用LoadBMP ("Data/Tim.bmp")替换它。

```
if (TextureImage[0]=LoadBMP("Data/Tim.bmp"))      // 载入位图
```

接着在InitGL()函数的尾部return TRUE之前，添加如下的代码。

```
glPolygonMode( GL_BACK, GL_FILL );          // 后表面完全填充
glPolygonMode( GL_FRONT, GL_LINE );          // 前表面使用线条绘制
```

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（译者：或者叫做后表面吧）。相反，多边形的正面（译者：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。详情请参考红宝书（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。接着上面的代码并在return TRUE这一句之前，添加如下的几行。

```
// 沿X平面循环
for(int x=0; x<45; x++)
{
```

```
// 沿Y平面循环
for(int y=0; y<45; y++)
{
    // 向表面添加波浪效果
    points[x][y][0]=float((x/5.0f)-4.5f);
    points[x][y][1]=float((y/5.0f)-4.5f);
    points[x][y][2]=float(sin((((x/5.0f)*40.0f)/360.0f)*3.141592654*2.0f));
}
}
```

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”（译者：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sine函数计算的结果。Sin()函数需要一个弧度参变量。将float\_x乘以40.0f，得到角度值。然后除以360.0f再乘以PI，乘以2，就转换为弧度了。

接着我将彻底重写DrawGLScene函数。

```
int DrawGLScene(GLvoid)           // 绘制我们的GL场景
{
    int x, y;                   // 循环变量
    float float_x, float_y, float_xb, float_yb; // 用来将旗形的波浪分割成很小的四边形
```

我们使用不同的变量来控制循环。下面的代码中大多数变量除了用来控制循环和存储临时变量之外并没有什么别的用处。

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除屏幕和深度缓冲
glLoadIdentity(); // 重置当前的模型观察矩阵
glTranslatef(0.0f,0.0f,-12.0f); // 移入屏幕12个单位
```

```

glRotatef(xrot,1.0f,0.0f,0.0f);           // 绕 X 轴旋转
glRotatef(yrot,0.0f,1.0f,0.0f);           // 绕 Y 轴旋转
glRotatef(zrot,0.0f,0.0f,1.0f);           // 绕 Z 轴旋转

glBindTexture(GL_TEXTURE_2D, texture[0]);    // 选择纹理

```

正如您所见，上面的代码和第六课的很类似，唯一的区别就是我将场景挪的离镜头更远了一些。

```

glBegin(GL_QUADS);                      // 四边形绘制开始
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)
    {

```

接着开始使用循环进行多边形绘制。这里使用整型可以避免我以前所用的int()强制类型转换。

```

float_x = float(x)/44.0f;               // 生成X浮点值
float_y = float(y)/44.0f;               // 生成Y浮点值
float_xb = float(x+1)/44.0f;             // X浮点值+0.0227f
float_yb = float(y+1)/44.0f;             // Y浮点值+0.0227f

```

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 $1/44 \times 1/44$ 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);        // 第一个纹理坐标 (左下角)
glVertex3f( points[x][y][0], points[x][y][1], points[x][y][2] );

glTexCoord2f( float_x, float_yb );       // 第二个纹理坐标 (左上角)
glVertex3f( points[x][y+1][0], points[x][y+1][1], points[x][y+1][2] );

```

```

glTexCoord2f( float_xb, float_yb ); // 第三个纹理坐标(右上角)
glVertex3f( points[x+1][y+1][0], points[x+1][y+1][1], points[x+1][y+1][2] );

glTexCoord2f( float_xb, float_y ); // 第四个纹理坐标(右下角)
glVertex3f( points[x+1][y][0], points[x+1][y][1], points[x+1][y][2] );
}

glEnd(); // 四边形绘制结束

```

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。

```

if( wiggle_count == 2 ) // 用来降低波浪速度(每隔2帧一次)
{

```

每绘制两次场景，循环一次sine值，以产生运动效果。

```

for( y = 0; y < 45; y++ ) // 沿Y平面循环
{
    hold=points[0][y][2]; // 存储当前左侧波浪值
    for( x = 0; x < 44; x++ ) // 沿X平面循环
    {
        // 当前波浪值等于其右侧的波浪值
        points[x][y][2] = points[x+1][y][2];
    }
    points[44][y][2]=hold; // 刚才的值成为最左侧的波浪值
}
wiggle_count = 0; // 计数器清零
}
wiggle_count++; // 计数器加一

```

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+=0.3f;           // X 轴旋转  
yrot+=0.2f;           // Y 轴旋转  
zrot+=0.4f;           // Z 轴旋转  
  
return TRUE;           // 返回  
}
```

标准的NeHe旋转增量。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。

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

#### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自



己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第10课

第12课 &gt;



## 第12课



显示列表:

想知道如何加速你的OpenGL程序么？这一课将告诉你如何使用OpenGL的显示列表，它通过预编译OpenGL命令来加速你的程序，并可以为你省去很多重复的代码。

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？用现实列表，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。给现实列表一个名字，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。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部构成，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。

下面设置变量。首先是存储纹理的变量，然后两个新的变量用于显示列表。这些变量是指向内存中显示列表的指针。命名为box和top。

然后用两个变量xloop,yloop表示屏幕上立方体的位置，两个变量xrot，yrot表示立方体的旋转。

```
GLuint box;           // 保存盒子的显示列表  
GLuint top;          // 保存盒子顶部的显示列表  
GLuint xloop;         // X轴循环变量  
GLuint yloop;         // Y轴循环变量
```

接下来建立两个颜色数组

```
static GLfloat boxcol[5][3]=           // 盒子的颜色数组  
{  
    // 亮:红，橙，黄，绿，蓝  
    {1.0f,0.0f,0.0f},{1.0f,0.5f,0.0f},{1.0f,1.0f,0.0f},{0.0f,1.0f,0.0f},{0.0f,1.0f,1.0f}  
};  
  
static GLfloat topcol[5][3]=           // 顶部的颜色数组  
{  
    // 暗:红，橙，黄，绿，蓝  
    {.5f,0.0f,0.0f},{0.5f,0.25f,0.0f},{0.5f,0.5f,0.0f},{0.0f,0.5f,0.0f},{0.0f,0.5f,0.5f}  
};
```

现在正式开始建立显示列表。你可能注意到了，所有创造盒子的代码都在第一个显示列表里，所有创造顶部的代码都在另一个列表里。我会努力解释这些细节。

```
GLvoid BuildLists() // 创建盒子的显示列表
{
```

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。

```
box=glGenLists(2); // 创建两个显示列表的名称
```

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。

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。而如果他已经编译成了.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。

```
glNewList(box,GL_COMPILE); // 创建第一个显示列表
```

下面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。

你可以在glNewList()和glEndList()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); // 开始绘制四边形  
    // 底面  
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);  
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);  
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);  
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);  
    // 前面  
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);  
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);  
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);  
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);  
    // 后面  
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);  
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);  
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);  
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);  
    // 右面  
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);  
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);  
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);  
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);  
    // 左面  
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);  
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);  
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);  
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);  
glEnd(); // 四边形绘制结束
```

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

```
glEndList(); // 第一个显示列表结束
```

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”。

```
top=box+1; // 第二个显示列表的名称
```

现在我们知道了第二个显示列表的指针，我们可以建立它了。

```
glNewList(top,GL_COMPILE); // 盒子顶部的显示列表
```

下面的代码画出盒子的顶部。

```
glBegin(GL_QUADS); // 开始绘制四边形  
// 上面  
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);  
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);  
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);  
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);  
glEnd(); // 结束绘制四边形
```

然后告诉OpenGL第二个显示列表建立完毕。

```
glEndList(); // 第二个显示列表创建完毕  
}
```

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。

```
if (TextureImage[0]=LoadBMP("Data/Cube.bmp"))
```

改变窗口大小的代码和第六课是一样的。

初始化的代码只有一点改变，加入了一行BuildList()。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。

```
BuildLists(); // 创建显示列表
```

接下来的三行使灯光有效。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。

```
glEnable(GL_LIGHT0); // 使用默认的0号灯  
glEnable(GL_LIGHTING); // 使用灯光  
glEnable(GL_COLOR_MATERIAL); // 使用颜色材质
```

现在到了绘制代码的地方了，我们还是和以前一样，以清除背景颜色为开始。

接着把纹理绑定到立方体，我可以把这些代码加入到显示列表中，但我还是把它留在了显示列表外边，这样我可以随便设置纹理。

```
int DrawGLScene(GLvoid) // 绘制操作开始
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除背景颜色
    glBindTexture(GL_TEXTURE_2D, texture[0]); // 选择纹理
```

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++) // 沿Y轴循环
{
```

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++) // 沿X轴循环
{
```

重置模型变化矩阵

```
glLoadIdentity(); // 重置模型变化矩阵
```

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。（原文有很多罗嗦的一段，相信大家的数学功底都不错，就不翻译了）

```
// 设置盒子的位置
glTranslatef(1.4f+(float(xloop)*2.8f)-(float(yloop)*1.4f),((6.0f-float(yloop))*2.4f)-7.0f,-20.0f);

glRotatef(45.0f-(2.0f*yloop)+xrot,1.0f,0.0f,0.0f);
glRotatef(45.0f+yrot,0.0f,1.0f,0.0f);
```

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); // 绘制盒子
```

然后用另外的颜色画顶部。搞定。

```
glColor3fv(topcol[yloop-1]); // 选择顶部颜色

glCallList(top); // 绘制顶部
}

}

return TRUE; // 成功返回
}
```

下面的代码是键盘控制的一些东西

```
SwapBuffers(hDC);           // 交换缓存  
if (keys[VK_LEFT])         // 左键是否按下  
{  
    yrot-=0.2f;             // 如果是，向左旋转  
}  
if (keys[VK_RIGHT])        // 右键是否按下  
{  
    yrot+=0.2f;             // 如果是向右旋转  
}  
if (keys[VK_UP])           // 上键是否按下  
{  
    xrot-=0.2f;             // 如果是向上旋转  
}  
if (keys[VK_DOWN])          // 下键是否按下  
{  
    xrot+=0.2f;             // 如果是向下旋转  
}
```

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

#### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容,大多都来自共享的



资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第11课

第13课 &gt;



## 第13课

图像字体:

Active OpenGL Text With NeHe - 33.56

这一课我们将创建一些基于2D图像的字体，它们可以缩放，但不能旋转，并且总是面向前方，但作为基本的显示来说，我想已经够了。

欢迎来到另一课教程，这次我将教你如何使用位图字体，也许你会对自己说：“在屏幕上显示文字有什么难的？”。但是你真正尝试过就会知道，它确实没那么容易。

当然，你可以载入一段美术程序，把文字写在一个图片上，再把这幅图片载入你的OpenGL程序中，打开混合选项，从而在屏幕上显示出文字。但是这种做法非常耗时。而且根据你选择的滤波类型，最终结果常常会显得很模糊，或者有很多马赛克。另外，除非你的图像包含一个Alpha通道，否则一旦绘制在屏幕上，那些文字就会不透明（与屏幕中的其它物体混合）。

如果你使用过记事本、微软的Word或者其它文字处理软件，你会注意到所有不同的字体都是可用的。这课就会教你如何在自己的OpenGL程序中使用和原样相同的字体。事实上，任何安装在你的计算机中的字体都可以使用在演示中（中文不行）。

使用位图字体比起使用图形字体（贴图）看起来不止强100倍。你可以随时改变显示在屏幕上的文字，而且用不着为它们逐个制作贴图。只需要将文字定位，再使用我最新的gl命令就可以在屏幕上显示文字了。

我尽可能试着将命令做的简单。你只需要敲入glPrint("Hello")。它是那么简单。不管怎

样，从这段长长的介绍就可以看出，我对这课教程是多么的满意。写这段代码大概花了我一个半小时，为什么这么长的时间呢？那是因为在使用位图字体方面完全没有可用的资料，除非你愿意使用MFC中的代码。为了使代码简单，我想，如果我把它全部重写为容易理解的C语言代码，那一定会好些：）

一个小注释，这段代码是专门针对Windows写的，它使用了Windows的wgl函数来创建字体，显然，Apple机系统有agl，X系统有glx来支持做同样事情的，不幸的是，我不能保证这些代码也是容易使用的。如果那位有能在屏幕上显示文字且独立于平台的代码，请告诉我，我将重写一个有关字体的教程。

我们从第一课的典型代码开始，添加上stdio.h头文件以便进行标准输入/输出操作，另外，stdarg.h头文件用来解析文字以及把变量转换为文字。最后加上math.h头文件，这样我们就可以使用SIN和COS函数在屏幕中移动文字了。

```
#include <stdarg.h> // 用来定义可变参数的头文件
```

另外，我们还要添加3个变量。base将保存我们创建的第一个显示列表的编号。每个字符都需要有自己的显示列表。例如，字符‘A’在显示列表中是65，‘B’是66，‘C’是67，等等。所以，字符‘A’应保存在显示列表中的base + 65这个位置。

然后添加两个计数器（cnt1 和 cnt2），它们采用不用的累加速度，通过SIN和COS函数来改变文字在屏幕上的位置。在屏幕上创造出一种看起来像是半随机的移动方式。同时，我们用这两个计数器来改变文字的颜色（后面会进一步解释）。

```
GLuint base; // 绘制字体的显示列表的开始位置
GLfloat cnt1; // 字体移动计数器1
GLfloat cnt2; // 字体移动计数器2
```

下面这段代码用来构建真实的字体，这也是最难写的一部分代码。‘HFONT font’告诉Windows我们将要使用一个Windows字体。Oldfont用来存放字体。

接下来我们在定义base的同时使用glGenLists(96)创建了一组共96个显示列表。

```
GLvoid BuildFont(GLvoid) // 创建位图字体
```

```

    HFONT font;           // 字体句柄
    HFONT oldfont;        // 旧的字体句柄

    base = glGenLists(96); // 创建96个显示列表
}

```

下面该有趣的部分了，我们将创建属于自己的字体。我们从指定字体的大小开始，你会注意到它是一个负数，我们通过加上一个负号来告诉Windows寻找一个基于CHARACTER高度的字体。如果我们使用一个正数，就是寻找一个与基于CELL的高度相匹配的字体。

```
font = CreateFont(-24, // 字体高度
```

然后我们指定每个单元的宽度，你会注意到我把它定义为0，这样，Windows就会使用默认值。如果你愿意的话，可以改变它的值，比如更宽一点，等等。

```
0, // 字体宽度
```

Angle Of Escapement会将字体旋转，它不是一个常用的属性，除了0, 90, 180, 270四个角度以外，由于字体本身要适应其看不见的方形边框，常常会显的裁切不正。MSDN帮助中解释Orientation Angle用于指定每个字的底边和显示设备的X轴之间的角度，每个单位是十分之一个角度，不幸的是我对这个没有概念。

```
0, // 字体的旋转角度 Angle Of Escapement
0, // 字体底线的旋转角度 Orientation Angle
```

字体重量是一个很重要的参数，你可以设置一个0 – 1000之间的值或使用一个已定义的值。FW\_DONTCARE是0, FW\_NORMAL是400, FW\_BOLD是700 and FW\_BLACK是900。还有许多预先定义的值，但是这四个的效果比较好。值越大，字体就越粗。

```
FW_BOLD,           // 字体的重量
```

Italic(斜体),Underline(下划线)和Strikeout (删除线) 可以是TRUE或FALSE。如果将 Underline设置为TRUE , 那么字体就会带有下划线 , 否则就没有 , 非常简单。

```
FALSE,           // 是否使用斜体
FALSE,           // 是否使用下划线
FALSE,           // 是否使用删除线
```

Character Set Identifier ( 字符集标识符 ) 用来描述你要使用的字符集 ( 内码 ) 类型。有太多需要说明的类型了。CHINESEBIG5\_CHARSET , GREEK\_CHARSET , RUSSIAN\_CHARSET , DEFAULT\_CHARSET , 等等。我使用的是ANSI , 尽管DEFAULT也是很好用的。

如果你有兴趣使用Webdings或Wingdings等字体 , 你必须使用SYMBOL\_CHARSET而不是ANSI\_CHARSET。

```
ANSI_CHARSET,    // 设置字符集
```

Output Precision ( 输出精度 ) 非常重要。它告诉Windows在有多种字符集的情况下使用哪类字符集。OUT\_TT\_PRECIS告诉Windows如果一个名字对应多种不同的选择字体 , 那么选择字体的TRUETYPE类型。Truetype字体通常看起来要好些 , 尤其是你把它们放大的时候。你也可以使用OUT\_TT\_ONLY\_PRECIS , 它将会一直尝试使用一种 TRUETYPE类型的字体

```
OUT_TT_PRECIS,   // 输出精度
```

裁剪精度是一种当字体落在裁剪范围之外时使用的剪辑类型 , 不用多说 , 只要把它设置为DEFAULT就可以了。

```
CLIP_DEFAULT_PRECIS, // 裁剪精度
```

输出质量非常重要。你可以使用PROOF , DRAFT , NONANTIALIASED , DEFAULT或ANTIALISED。

我们都知道 , ANTIALIASED字体看起来很好 , 将一种字体Antialiasing(反锯齿)可以实现 在Windows下打开字体平滑时同样的效果 , 它使任何东西看起来都要少些锯齿 , 也就是更平滑。

```
ANTIALIASED_QUALITY, // 输出质量
```

下面是Family和Pitch设置。Pitch属性有DEFAULT\_PITCH , FIXED\_PITCH和VARIABLE\_PITCH , Family有FF\_DECORATIVE,FF\_MODERN,FF\_ROMAN,FF\_SCRIPT,FF\_SWISS,FF\_DONT CARE.尝试一下这些值 , 你就会知道它们到底有什么功能。我把它们都设置为默认值。

```
FF_DONT CARE|DEFAULT_PITCH, // Family And Pitch
```

最后 , 是我们需要的字体的确切的名字。打开Microsoft Word或其它什么文字处理软件 , 点击字体下拉菜单 , 找一个你喜欢的字体。将 ‘ Courier New ’ 替换为你想用的字体的名字 , 你就可以使用它了。 ( 中文还不行 , 需要别的方法 )

```
"Courier New"); // 字体名称
```

现在 , 选择我们刚才创建的字体。Oldfont将指向被选择的对象。然后我们从第32个字符 ( 空格 ) 开始建立96个显示列表。如果你愿意 , 也可以建立所有256个字符 , 只要确保 使用glGenLists建立256个显示列表就可以了。然后我们将oldfont对象指针选入hDC并且删除font对象。

```

oldfont = (HFONT)SelectObject(hDC, font);           // 选择我们需要的字体
wglUseFontBitmaps(hDC, 32, 96, base);             // 创建96个显示列表，绘制从ASCII码为32-128的
                                                // 字符
SelectObject(hDC, oldfont);                      // 选择原来的字体
DeleteObject(font);                             // 删除字体
}

```

接下来的代码很简单。它在内存中从base开始删除96个显示列表。我不知道Windows是否会做这些工作，但还是保险为好。

```

GLvoid KillFont(GLvoid)                         // 删除显示列表
{
    glDeleteLists(base, 96);                     // 删除96个显示列表
}

```

下面就是我优异的GL文字程序了。你可以通过调用glPrint(“需要写的文字”)来调用这段代码。文字被存储在字符串 \* fmt中。

```

GLvoid glPrint(const char *fmt, ...)           // 自定义GL输出字体函数
{

```

下面的第一行创建了一个大小为256个字符的字符数组，里面保存我们想要的文字串。第二行创建了一个指向一个变量列表的指针。我们在传递字符串的同时也传递了这个变量列表。如果我们传递文本时也传递了变量，这个指针将指向它们。

```

char      text[256];          // 保存文字串
va_list   ap;                // 指向一个变量列表的指针

```

下面两行代码检查是否有需要显示的内容，如果什么也没有，fmt就等于空（NULL），屏幕上也就什么都没有。

```
if (fmt == NULL) // 如果无输入则返回
    return;
```

接下来三行代码将文字中的所有符号转换为它们的字符编号。最后，文字和转换的符号被存储在一个叫做text的字符串中。以后我会多解释一些有关字符的细节。

```
va_start(ap, fmt); // 分析可变参数
vsprintf(text, fmt, ap); // 把参数值写入字符串
va_end(ap); // 结束分析
```

然后我们将GL\_LIST\_BIT压入属性堆栈，它会防止glListBase影响到我们的程序中的其它显示列表。

glListBase(base-32)是一条有些难解释的命令。比如说要写字母‘A’，它的相应编号为65。如果没有glListBase(base-32)命令，OpenGL就不知道到哪去找这个字母。它会在显示列表中的第65个位置找它，但是，假如base的值等于1000，那么‘A’的实际存放位置就是1065了。所以通过base设置一个起点，OpenGL就知道到哪去找到正确的显示列表了。减去32是因为我们没有构造过前32个显示列表，那么就跳过它们好了。于是，我们不得不通过从base的值减去32来让OpenGL知道这一点。我希望这些有意义。

```
glPushAttrib(GL_LIST_BIT); // 把显示列表属性压入属性堆栈
glListBase(base - 32); // 设置显示列表的基础值
```

现在OpenGL知道字母的存放位置了，我们就可以让它在屏幕上显示文字了。glCallLists是一个很有趣的命令。它可以同时将多个显示列表的内容显示在屏幕上。

下面的代码做后续工作。首先，它告诉OpenGL我们将要在屏幕上显示出显示列表中的内容。Strlen(text)函数用来计算我们将要显示在屏幕上的文字的长度。然后，OpenGL需要知道我们允许发送给它的列表的最大值。我们不能发送长度大于255的字符串。这个字符列表的参数被当作一个无符号字符数组处理，它们的值都介于0到255之间。最后，我们通过传递text(它指向我们的字符串)来告诉OpenGL显示的内容。

也许你想知道为什么字符不会彼此重叠堆积在一起。那时因为每个字符的显示列表都知道字符的右边缘在那里，在写完一个字符后，OpenGL自动移动到刚写过的字符的右边，在写下一个字或画下一个物体时就会从GL移动到的最后的位置开始，也就是最后一个字符的右边。

最后，我们将GL\_LIST\_BIT属性弹出堆栈，将GL恢复到我们使用glListBase(base-32)设置base那时的状态。

```
glCallLists(strlen(text), GL_UNSIGNED_BYTE, text);      // 调用显示列表绘制字符串  
glPopAttrib();                                         // 弹出属性堆栈  
}
```

在初始化代码中唯一的变化就是BuildFont()。它调用前面的代码来创建字体，然后OpenGL就可以使用这个字体了。

```
BuildFont();                                         // 创建字体
```

下面就是画图的代码了。我们从清除屏幕和深度缓存开始。我们调用glLoadIdentity()来重置所有东西。然后我们将坐标系向屏幕里移动一个单位。如果不移动的话无法显示出文字。当你使用透视投影而不是ortho投影的时候位图字体表现的更好。由于ortho看起来不好，所以我用透视投影，并移动坐标系。。

你会注意到如果把坐标系在屏幕里放的更深远，字体并不会像你想象的那样缩小，只是你可以在控制文字位置时有更多的选择。如果你将坐标系移入屏幕一个单位，你就可以在X轴上-0.5到+0.5的范围内调整文字的位置。如果深入10个单位的话，移动范围就从-5到+5。它给了你更多的选择来替代使用小数指定文字的精确位置。什么都不能改变文字的大小，即使是调用glScale(x,y,z)函数。如果你想改变字体的大小，只能在创建它的时候改变它。

```
int DrawGLScene(GLvoid) // 此过程中包括所有的绘制代码
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除屏幕及深度缓存
    glLoadIdentity(); // 重置当前的模型观察矩阵
    glTranslatef(0.0f,0.0f,-1.0f); // 移入屏幕一个单位
```

下面我们使用一些奇妙的数学方法来产生颜色变化脉冲。如果你不懂我在做什么你也不必担心。我喜欢利用教多的变量和教简单的方法来达到我的目的。

这样，我使用那两个用来改变文字在屏幕上位置的计数器来改变红、绿、蓝这些颜色。红色值使用COS和计数器1在-1.0到1.0之间变化。绿色值使用SIN和计数器2也在-1.0到1.0之间变化。蓝色值使用COS和计数器1和2在0.5到1.5之间变化。于是，蓝色值就永远不会等于0，文字的颜色也永远不会消失。笨办法，但很管用。

```
// 根据字体位置设置颜色
	glColor3f(1.0f*float(cos(cnt1)),1.0f*float(sin(cnt2)),1.0f-0.5f*float(cos(cnt1+cnt2)));
```

下面是一个新命令。GLRasterPos2f(x,y)用于在屏幕上定位位图字体。屏幕的中心依然是(0,0)，注意，这里没有Z轴位置。位图字体只使用X轴(左/右)和Y轴(上/下)。因为我们将坐标系移入屏幕一个单位，往左最大值为-0.5，往右最大值为+0.5。你会注意到我在X轴上向左移动了0.45个像素。它将文字移到屏幕的中心位置。否则，因为文字的起点就是屏幕的中心，会造成文字整体偏右。

计算文字位置的算法与设置文字颜色的算法差不多。它将文字在X轴的-0.50到-0.40的范围内移动(记住，我们从起点就减了0.45)，这就保证文字始终能显示在屏幕内。由于使用COS和计数器1，所以文字左右摆动，使用SIN和计数器2在Y轴的-0.35到0.35范围内移动。

```
// 设置光栅化位置，即字体的位置
glRasterPos2f(-0.45f+0.05f*float(cos(cnt1)), 0.35f*float(sin(cnt2)));
```

现在轮到我最满意的部分了。将真正的文字写到屏幕上。我试着把它做的非常简单，而且非常友好，便于使用。你会注意到它看起来像调用一个OpenGL的函数，有点类似C语言中的输出语句的风格。在屏幕上输出文字只需要调用glPrint("你想写的文字").它很容易。文字将精确的显示在屏幕上你指定的位置。

Shawn T.发给我修改过的代码允许glPrint传递变量到屏幕。这意味着你可以增加一个计数器，并且在屏幕上显示出这个计数器的值，它是这样工作的。。。在下一行你看到：要显示的普通文字，然后有一个空格，一个破折号，一个空格，然后是一个“符号”(%7.2f)(C语言中的输出格式控制字).现在你会看着%7.2说这是什么意思。它其实很简单，%是一个记号，表示不要把7.2f本身显示在屏幕上，因为它代表一个变量。7表示小数点左边最多有7位数字。然后是小数部分，小数点右边的2表示小数点右边最多保留两位小数。最后，f表示我们想要显示的数字类型为浮点型。我们想在屏幕上显示计数器1的值。比如，计数器1的值为300.12345f，那么在屏幕上显示的数字就是300.12，小数部分的3,4,5会舍去。因为我们只需要显示小数点后面两位数字。

我知道如果你是一个有经验的C程序员，这是个很基础的问题。不过也许也有人没有用过printf函数。如果你想了解更多的字符，那就买本书或者查阅MSDN。

```
glPrint("Active OpenGL Text With NeHe - %7.2f", cnt1); // 输出文字到屏幕
```

最后一件事就是以不同的速率增加计数器的值来产生颜色脉冲并且移动文字。

```
cnt1+=0.051f;           // 增加计数器值  
cnt2+=0.005f;           // 增加计数器值  
return TRUE;             // 继续运行  
}
```

最后，如下所示，就是增加在KillGLWindow()函数中增加KillFont()函数，这很重要，它在我们退出程序之前做清理工作。

```
KillFont();           // 删除字体
```

好了，用于使用位图字体的所有一切都在你的OpenGL程序中了。我在网上寻找过与这篇教程相似的文章，但没有找到。或许我的网站是第一个涉及这个主题的C代码的网站吧。不管怎样，享用这篇教程，快乐编码！

#### 版权与使用声明：

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

#### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我



都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第12课

第14课 >



## 第14课



图形字体:

在一课我们将教你绘制3D的图形字体，它们可像一般的3D模型一样被变换。

这节课继续上一节课课的内容。在第13课我们学习了如何使用位图字体，这节课，我们将学习如何使用轮廓字体。

创建轮廓字体的方法类似于在第13课中我们创建位图字体的方法。但是，轮廓字体看起来要酷100倍！你可以指定轮廓字体的大小。轮廓字体可以在屏幕中以3D方式运动，而且轮廓字体还可以有一定的厚度！而不是平面的2D字符。使用轮廓字体，你可以将你的计算机中的任何字体转换为OpenGL中的3D字体，加上合适的法线，在有光照的时候，字符就会被很好的照亮了。

一个小注释，这段代码是专门针对Windows写的，它使用了Windows的wgl函数来创建字体，显然，Apple机系统有agl，X系统有glx来支持做同样事情的，不幸的是，我不能保证这些代码也是容易使用的。如果哪位有能在屏幕上显示文字且独立于平台的代码，请告诉我，我将重写一个有关字体的教程。

我们从第一课的典型代码开始，添加上stdio.h头文件以便进行标准输入/输出操作，另外，stdarg.h头文件用来解析文字以及把变量转换为文字。最后加上math.h头文件，这样我们就可以使用SIN和COS函数在屏幕中移动文字了。

另外，我们还要添加2个变量。base将保存我们创建的第一个显示列表的编号。每个字符都需要有自己的显示列表。例如，字符‘A’在显示列表中是65，‘B’是66，‘C’是67，等等。所以，字符‘A’应保存在显示列表中的base + 65这个位置。

我们再添加一个叫做rot的变量。用它配合SIN和COS函数在屏幕上旋转文字。我们同时用它来改变文字的颜色。

```
GLuint base;           // 绘制字体的显示列表的开始位置
GLfloat rot;          // 旋转字体
```

GLYPHMETRICSFLOAT gmf[256]用来保存256个轮廓字体显示列表中对应的每一个列表的位置和方向的信息。我们通过gmf[num]来选择字母。num就是我们想要了解的显示列表的编号。在稍后的代码中，我将说明如何如何检查每个字符的宽度，以便自动将文字定位在屏幕中心。切记，每个字符的宽度可以不相同。Glyphmetrics会大大简化我们的工作。

```
GLYPHMETRICSFLOAT gmf[256]; // 记录256个字符的信息
```

下面这段用来构建真正的字体的代码类似于我们创建位图字体的方法。和13课一样，只是使用wglUseFontOutlines函数替换wglUseFontBitmaps函数。

```
base = glGenLists(256);           // 创建256个显示列表
wglUseFontOutlines( hDC,           // 设置当前窗口设备描述表的句柄
                    0,             // 用于创建显示列表字体的第一个字符的ASCII值
                    255,            // 字符数
                    base,           // 第一个显示列表的名称
```

That's not all however. We then set the deviation level. The closer to 0.0f, the smooth the font will look. After we set the deviation, we get to set the font thickness. This describes how thick the font is on the Z axis. 0.0f will produce a flat 2D looking font and 1.0f will produce a font with some depth.

The parameter WGL\_FONT\_POLYGONS tells OpenGL to create a solid font using polygons. If we use WGL\_FONT\_LINES instead, the font will be wireframe (made of lines). It's also important to note that if you use GL\_FONT\_LINES, normals will not be generated so lighting will not work properly.

The last parameter gmf points to the address buffer for the display list data.

```
    0.0f,           // 字体的光滑度，越小越光滑，0.0为最光滑的状态
    0.2f,           // 在z方向突出的距离
    WGL_FONT_POLYGONS,   // 使用多边形来生成字符，每个顶点具有独
                         立的法线
    gmf);          //一个接收字形度量数据的数据的地址，每个数组元素用它对
                     的显示列表字符的数据填充
}
```

The following code is pretty simple. It deletes the 256 display lists from memory starting at the first list specified by base. I'm not sure if Windows would do this for you, but it's better to be safe than sorry :)

```
GLvoid KillFont(GLvoid)           // 删除显示列表
{
    glDeleteLists(base, 256);      // 删除256个显示列表
}
```

下面就是我优异的GL文字程序了。你可以通过调用glPrint(“需要写得文字”)来调用这段代码。文字被存储在字符串text[]中。

```
GLvoid glPrint(const char *fmt, ...) // 自定义GL输出字体函数
{
}
```

下面的第一行定义了一个叫做length的变量。我们使用这个变量来查询字符串的长度。  
第二行创建了一个大小为256个字符的字符数组，里面保存我们想要的文字串。第三行  
创建了一个指向一个变量列表的指针，我们在传递字符串的同时也传递了这个变量列  
表。如果我们传递文字时也传递了变量，这个指针将指向它们。

```
float    length=0;           // 查询字符串的长度
char     text[256];          // 保存我们想要的文字串
va_list  ap;                // 指向一个变量列表的指针
```

下面两行代码检查是否有需要显示的内容，如果什么也没有，屏幕上也就什么都没有。

```
if (fmt == NULL)           // 如果无输入则返回
    return;
```

接下来三行代码将文字中的所有符号转换为它们的字符编号。最后，文字和转换的符号  
被存储在一个叫做“text”的字符串中。以后我会多解释一些有关字符的细节。

```
va_start(ap, fmt);         // 分析可变参数
vsprintf(text, fmt, ap);   // 把参数值写入字符串
va_end(ap);                // 结束分析
```

感谢Jim Williams对下面一段代码的建议。以前我是用手工将文字置于中心的，而他的办法要好的多。

我们从一个循环开始，它将逐个检查文本中的字符。我们通过strlen(text)得到文本的长度。设置好了循环以后，我们将通过加上每个字符的长度来增加length的值。当循环结束以后，被保存在length中的值就是整个字符串的长度。所以，如果我们要写的是“hello”，假设每个字符的长度都为10个单位，我们先给length的值加上第一个字母的长度10。然后，我们检查第二个字母的长度，它的长度也是10，所以length就变成 $10 + 10 = 20$ 。当我们检查完所有5个字母以后，length的值就会等于50 ( $5 * 10$ )。

给出我们每个字符的长度的代码是gmf[text[loop]].gmfCellIncX。记住，gmf存储了我们每个显示列表的信息。如果loop等于0，text[loop]就是我们的字符串中的第一个字符。如果loop等于1，text[loop]就是我们的字符串中的第二个字符。gmfCellIncX告诉我们被选择的字符的长度。GmfCellIncX表示显示位置从已绘制上的上一个字符向右移动的真正距离，这样，字符之间就不会重叠在一起。同时，这个距离就是我们想得到的字符的宽度。你还可以通过gmfCellIncY命令来得到字符的高度。如果你是在垂直方向绘制文本而不是在水平方向时，这会很方便。

```
for (unsigned int loop=0;loop<(strlen(text));loop++) // 查找整个字符串的长度
{
    length+=gmf[text[loop]].gmfCellIncX;
}
```

最后我们取出计算后得到的length，并把它变成负数（因为我们要将文本从屏幕中心左移从而把整个文本置于屏幕中间）。然后我们把length除以2。我们并不想移动整个文本的长度，只需要一半！

```
glTranslatef(-length/2,0.0f,0.0f); // 把字符串置于最左边
```

然后我们将GL\_LIST\_BIT压入属性堆栈，它会防止glListBase影响到我们的程序中的其它显示列表

```
glPushAttrib(GL_LIST_BIT); // 把显示列表属性压入属性堆栈
glListBase(base); // 设置显示列表的基础值为0
```

现在OpenGL知道字符的存放位置了，我们就可以让它在屏幕上显示文字了。glCallLists会调用多个显示列表从而把整个文字的内容同时显示在屏幕上。

下面的代码做后续工作。首先，它告诉OpenGL我们将要在屏幕上显示出显示列表中的内容。Strlen(text)函数用来计算我们将要显示在屏幕上的文字的长度。然后，OpenGL需要知道我们允许发送给它的列表的最大值。我们依然不能发送长度大于255的字符串。所以我们使用UNSIGNED\_BYTE。（用0 - 255来表示我们需要的字符）。最后，我们通过传递字符串文字告诉OpenGL显示什么内容。

也许你想知道为什么字符不会彼此重叠堆积在一起。那时因为每个字符的显示列表都知道字符的右边缘在那里，在写完一个字符后，OpenGL自动移动到刚写过的字符的右边，在写下一个字或画下一个物体时就会从GL移动到的最后的位置开始，也就是最后一个字符的右边。

最后，我们将GL\_LIST\_BIT属性弹出堆栈，将GL恢复到我们使用glListBase(base)设置base之前的状态。

```
glCallLists(strlen(text), GL_UNSIGNED_BYTE, text); // 调用显示列表绘制字符串
glPopAttrib(); // 弹出属性堆栈
}
```

下面就是画图的代码了。我们从清除屏幕和深度缓存开始。我们调用glLoadIdentity()来重置所有东西。然后我们将坐标系向屏幕里移动十个单位。轮廓字体在透视图模式下表现非常好。你将文字移入屏幕越深，文字看起来就更小。文字离你越近，它看起来就更大。

也可以使用glScalef(x,y,z)命令来操作轮廓字体。如果你想把字体放大两倍，可以使用glScalef(1.0f,2.0f,1.0f). 2.0f作用在y轴，它告诉OpenGL将显示列表的高度绘制为原来的两倍。如果2.0f作用在x轴，那么文本的宽度将变成原来的两倍。

```
int DrawGLScene(GLvoid) // 此过程中包括所有的绘制代码
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除屏幕及深度缓存
    glLoadIdentity(); // 重置当前的模型观察矩阵
    glTranslatef(0.0f,0.0f,-10.0f); // 移入屏幕一个单位
```

在向屏幕里移动以后，我们希望文本能旋转起来。下面3行代码用来在3个轴上旋转屏幕。我将rot乘以不同的数，以便每个方向上的旋转速度不同。

```
glRotatef(rot,1.0f,0.0f,0.0f);           // 沿X轴旋转
glRotatef(rot*1.5f,0.0f,1.0f,0.0f);       // 沿Y轴旋转
glRotatef(rot*1.4f,0.0f,0.0f,1.0f);       // 沿Z轴旋转
```

下面是令人兴奋的颜色循环了。照常，我们使用唯一递增的变量（rot）。颜色通过使用COS和SIN来循环变化。我将rot除以不同的数，这样每种颜色会以不同的速度递增。最终的效果非常好。

```
// 根据字体位置设置颜色
glColor3f(1.0f*float(cos(rot/20.0f)),1.0f*float(sin(rot/25.0f)),1.0f-0.5f*float(cos(rot/17.0f)));
```

我最喜欢的部分，将文字写到屏幕上。我使用同将位图字体写到屏幕上相同的函数。将文字写在屏幕上，所有你要做的就是glPrint(“你想写的文字”)。很简单。

在下面的代码中，我们要写的是NeHe，空格，破折号，空格，然后是rot的值除以50后的结果（为了减慢计数器）。如果这个数大于999.99,左边第四个数将被去掉（我们要求只显示小数点左边3位数字）。只显示小数点右边的两位数字。

```
glPrint("NeHe - %3.2f",rot/50);          // 输出文字到屏幕
```

然后增大旋转变量从而改变颜色并旋转文字。

```
rot+=0.5f;                                // 增加旋转变量
return TRUE;                               // 成功返回
```

在这节课结束的时候，你应该已经学会在你的OpenGL程序中使用轮廓字体了。就像第13课，我曾在网上寻找一篇与这一课相似的教程，但是也没有找到。或许我的网站是第一个涉及这个主题同时又把它解释的简单易懂的C代码的网站吧。享用这篇教程，快乐编码！

### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照

顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第13课

第15课 >



## 第15课



图形字体的纹理映射:

这一课，我们将在上一课的基础上创建带有纹理的字体，它真的很简单。

在发布了前两篇关于位图字体和轮廓字体的教程以后，我收到很多邮件，很多读者都想知道如何才能给字体赋予纹理贴图。你可以使用自动纹理坐标生成器。它会为字体上的每一个多边形生成纹理坐标。

一个小注释，这段代码是专门针对Windows写的，它使用了Windows的wgl函数来创建字体，显然，Apple机系统有agl，X系统有glx来支持做同样事情的，不幸的是，我不能保证这些代码也是容易使用的。如果哪位有能在屏幕上显示文字且独立于平台的代码，请告诉我，我将重写一个有关字体的教程。

我们将使用第14课的代码来创作纹理字体的演示。如果程序中哪部分的代码有变化，我会重写那部分的所有代码以便看出我做的改动。

我们还要添加一个叫做texture[]的整型变量。它用于保存纹理。后面3行是第14课中的代码，本课不做改动。

下面的部分做了一些小改动。我打算在这课使用wingdings字体来显示一个海盗旗（骷髅头和十字骨头）的标志。如果你想显示文字的话，就不用改动第14课中的代码了，也可以选择另一种字体。

有些人想知道如何使用wingdings字体，这也是我不用标准字体的一个原因。wingdings是一种符号字体，使用它时需要做一些改动。告诉Windows使用wingdings字体并不太简单。如果你把字体的名字改为wingdings，你会注意到字体其实并没有选到。你必须告诉Windows这种字体是一种符号字体而不是一种标准字符字体。后面会继续解释。

```
GLvoid BuildFont(GLvoid) // 创建位图字体
{
    GLYPHMETRICSFLOAT gmf[256]; // 记录256个字符的信息
    HFONT font; // 字体句柄

    base = glGenLists(256); // 创建256个显示列表
    font = CreateFont(-12, // 字体高度
                      0, // 字体宽度
                      0, // 字体的旋转角度 Angle Of Escapement
                      0, // 字体底线的旋转角度 Orientation Angle
                      FW_BOLD, // 字体的重量
                      FALSE, // 是否使用斜体
                      FALSE, // 是否使用下划线
                      FALSE); // 是否使用删除线
}
```

这就是有魔力的那一行！不使用第14课中的ANSI\_CHARSET，我们将使用SYMBOL\_CHARSET。这会告诉Windows我们创建的字体并不是由标准字符组成的典型字体。所谓符号字体通常是由一些小图片（符号）组成的。如果你忘了改变这行，wingdings,webdings以及你想用的其它符号字体就不会工作。

```
SYMBOL_CHARSET, // 设置字符集
```

下面几行没有变化。

```

    OUT_TT_PRECIS,           // 输出精度
    CLIP_DEFAULT_PRECIS,     // 裁剪精度
    ANTIALIASED_QUALITY,    // 输出质量
    FF_DONTCARE|DEFAULT_PITCH, // Family And Pitch

```

既然我们已经选择了符号字符集标识符，我们就可以选择wingdings字体了

```
"Wingdings");           // 字体名称
```

剩下几行代码没有变化

```

SelectObject(hDC, font);           // 选择创建的字体

wglUseFontOutlines( hDC,           // 设置当前窗口设备描述表的句柄
                    0,             // 用于创建显示列表字体的第一个字符的ASCII值
                    255,            // 字符数
                    base,           // 第一个显示列表的名称

```

我们允许有更多的误差，这意味着GL不会严格的遵守字体的轮廓线。如果你把误差设置为0.0f，你就会发现严格地在曲面上贴图存在一些问题。但是如果你允许一定的误差，很多问题都可以避免。

```
0.1f,           // 字体的光滑度，越小越光滑，0.0为最光滑的状态
```

下面三行代码还是相同的

```

        0.2f,           // 在z方向突出的距离
WGL_FONT_POLYGONS,      // 使用多边形来生成字符，每个顶点具有独
立的法线
gmf);          // 一个接收字形度量数据的数据的数组地址，每个数组元素用它对
应的显示列表字符的数据填充
}

```

在ReSizeGLScene()函数之前，我们要加上下面一段代码来读取纹理。你可能会认得这些前几课中的代码。我们创建一个保存位图的地方，读取位图，告诉Windows生成一个纹理，并把它保存在texture[0]中。

我们创建一种细化纹理（mipmapped texture），这样会看起来好些。纹理的名字叫做lights.bmp。

```
if (TextureImage[0]=LoadBMP("Data/Lights.bmp"))           // 载入位图
```

下面四行代码将为我们绘制在屏幕上的任何物体自动生成纹理坐标。函数glTexGen非常强大，而且复杂，如果要完全讲清楚它的数学原理需要再写一篇教程。不过，你只要知道GL\_S和GL\_T是纹理坐标就可以了。默认状态下，它被设置为提取物体此刻在屏幕上的x坐标和y坐标，并把它们转换为顶点坐标。你会发现到物体在z平面没有纹理，只显示一些斑纹。正面和反面都被赋予了纹理，这些都是由glTexGen函数产生的。（X(GL\_S)用于从左到右映射纹理，Y(GL\_T)用于从上到下映射纹理。）

GL\_TEXTURE\_GEN\_MODE允许我们选择我们想在S和T纹理坐标上使用的纹理映射模式。你有3种选择：

GL\_EYE\_LINEAR - 纹理会固定在屏幕上。它永远不会移动。物体将被赋予处于它通过的地区的那一块纹理。

GL\_OBJECT\_LINEAR - 这种就是我们使用的模式。纹理被固定于在屏幕上运动的物体上。

GL\_SPHERE\_MAP - 每个人都喜欢。创建一种有金属质感的物体。

```
// 设置纹理映射模式
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
```

```
glTexGen(GL_T, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
glEnable(GL_TEXTURE_GEN_S);           // 使用自动生成纹理
glEnable(GL_TEXTURE_GEN_T);
```

在InitGL()的最后有几行新代码。BuildFont()被放到了读取纹理的代码之后。glEnable(GL\_COLOR\_MATERIAL)这行被删掉了，如果你想使用glColor3f(r,g,b)来改变纹理的颜色，那么就把glEnable(GL\_COLOR\_MATERIAL)这行重新加到这部分代码中。

```
int InitGL(GLvoid)                                // 此处开始对OpenGL进行所有设置
{
    if (!LoadGLTextures())                         // 载入纹理
    {
        return FALSE;                             // 失败则返回
    }
    BuildFont();                                // 创建字体显示列表

    glShadeModel(GL_SMOOTH);                     // 启用阴影平滑
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);         // 黑色背景
    glClearDepth(1.0f);                          // 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
    glEnable(GL_LIGHT0);                         // 使用第0号灯
    glEnable(GL_LIGHTING);                       // 使用光照
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // 告诉系统对透视进行修正
```

启动2D纹理映射，并选择第一个纹理。这样就把第一个纹理映射到我们绘制在屏幕上的3D物体上了。如果你想加入更多的操作，可以按自己的意愿启动或禁用纹理映射。

```
glEnable(GL_TEXTURE_2D);                         // 使用二维纹理
glBindTexture(GL_TEXTURE_2D, texture[0]);          // 选择使用的纹理
return TRUE;                                    // 初始化成功
}
```

重置大小的代码没有变化，但DrawGLScene这部分代码有变化。

这里是第一处变动。我们打算使用COS和SIN让物体绕着屏幕旋转而不是把它固定在屏幕中间。我们将把物体向屏幕里移动3个单位。在x轴，我们将移动范围限制在-1.1到+1.1之间。我们使用rot变量来控制左右移动。我们把上下移动的范围限制在+0.8到-0.8之间。同样使用rot变量来控制上下移动（最好充分利用你的变量）。

```
// 设置字体的位置  
glTranslatef(1.1f*float(cos(rot/16.0f)),0.8f*float(sin(rot/20.0f)), -3.0f);
```

下面做常规的旋转。这会使符号在X，Y和Z轴旋转。

```
glRotatef(rot,1.0f,0.0f,0.0f);           // 沿X轴旋转  
glRotatef(rot*1.2f,0.0f,1.0f,0.0f);       // 沿Y轴旋转  
glRotatef(rot*1.4f,0.0f,0.0f,1.0f);       // 沿Z轴旋转
```

我们将物体相对观察点向左向下移动一点，以便于把符号定位在每个轴的中心。否则，当我们旋转它的时候，看起来就不像是在围绕它自己的中心在旋转。-0.35只是一个能让符号正确显示的数。我也试过一些其它数，因为我不知道这种字体的宽度是多少，可以适情况作出调整。我不知道为什么这种字体没有一个中心。

```
glTranslatef(-0.35f,-0.35f,0.1f);          // 移动到可以显示的位置
```

最后，我们绘制海盗旗的符号，然后增加rot变量，从而使这个符号在屏幕中旋转和移动。如果你不知道我是如何从字母‘N’中得到海盗旗符号的，那就打开Microsoft Word或是写字板。在字体下拉菜单中选择Wingdings字体。输入大写字母‘N’，就会显示出海盗旗符号了。

```
glPrint("N");
rot+=0.1f;                                // 绘制海盗旗符号
                                                // 增加旋转变量
```

最后要做的事就是在KillGLWindow()的最后添加KillFont()函数，如下所示。添加这行代码很重要。它将在我们退出程序之前做清理工作。

尽管我没有讲的细致入微，但我想你应该很好的理解了如何让OpenGL为你生成纹理坐标。在给你的字体或者是同类物体赋予纹理映射时，应该没有问题了，而且只需要改变两行代码，你就可以启用球体映射了，它的效果简直酷毙了！

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

#### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

#### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢



积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第14课

第16课 >



## 第16课



雾:

这一课是基于第7课的代码的，你将学会三种不同的雾的计算方法，以及怎样设置雾的颜色和雾的范围。

这篇教程由Chris Aliotta编写。

你想给你的OpenGL程序添加雾效?我将在这篇教程中教你如何去做。这是我第一次写教程，我也只是OpenGL/C++编程的新手，所以如果你发现了什么错误请告诉我而不是叱责我。本课的代码基于第七课的代码编写。

Data Setup:

我们将从设置保存雾的信息的变量开始。变量fogMode用来保存三种类型的雾:GL\_EXP, GL\_EXP2和GL\_LINEAR。我将在稍后解释这三种类型的区别。这些变量位于程序开头 GLuint texture[3]这行后。变量fogfilter将用来表示我们使用的是哪种雾类型。变量fogColor保存雾的颜色。在程序的顶部我还加了一个布尔类型的变量gp用来记录'g'键是否被按下。

```
bool gp; // G健是否按下  
GLuint filter; // 使用哪一个纹理过滤器
```

```

GLuint fogMode[] = { GL_EXP, GL_EXP2, GL_LINEAR };           // 雾气的模式
GLuint fogfilter = 0;                                     // 使用哪一种雾气
GLfloat fogColor[4] = {0.5f, 0.5f, 0.5f, 1.0f};          // 雾的颜色设为白色

```

现在我们已经设置了变量，接下来我们来看InitGL函数。为了获得更好的效果，glClearColor()这行已经被修改为将屏幕清为同雾相同的颜色。使用雾效只需很少的代码。总之你将发现这很容易。

```
glClearColor(0.5f, 0.5f, 0.5f, 1.0f); // 设置背景的颜色为雾气的颜色
```

```

glFogi(GL_FOG_MODE, fogMode[fogfilter]); // 设置雾气的模式
glFogfv(GL_FOG_COLOR, fogColor);        // 设置雾的颜色
glFogf(GL_FOG_DENSITY, 0.35f);          // 设置雾的密度
glHint(GL_FOG_HINT, GL_DONT_CARE);      // 设置系统如何计算雾气
glFogf(GL_FOG_START, 1.0f);             // 雾气的开始位置
glFogf(GL_FOG_END, 5.0f);               // 雾气的结束位置
glEnable(GL_FOG);                      // 使用雾气

```

让我们先看看这段代码的头三行。第一行glEnable(GL\_FOG);(?这应该是最后一行)不用再解释了吧，主要是初始化雾效。

第二行glFogi(GL\_FOG\_MODE, fogMode[fogfilter]);建立雾的过滤模式。之前我们声明了数组fogMode，它保存了值GL\_EXP, GL\_EXP2, and GL\_LINEAR。现在是使用他们的时候了。我来解释它们（具体见红皮书，其实这是计算雾效混合因子的三种方式）：

GL\_EXP - 充满整个屏幕的基本渲染的雾。它能在较老的PC上工作，因此并不是特别像雾。

GL\_EXP2 - 比GL\_EXP更进一步。它也是充满整个屏幕，但它使屏幕看起来更有深度。

GL\_LINEAR - 最好的渲染模式。物体淡入淡出的效果更自然。

第三行glFogfv(GL\_FOG\_COLOR, fogcolor);设置雾的颜色。之前我们已将变量fogcolor设为(0.5f, 0.5f, 0.5f, 1.0f)，这是一个很棒的灰色。

接下来我们看看最后的四行。glFogf(GL\_FOG\_DENSITY, 0.35f);这行设置雾的密度。增加数字会让雾更密，减少它则雾更稀。

glHint(GL\_FOG\_HINT, GL\_DONT\_CARE);设置修正。我使用了GL\_DONT\_CARE因为我不关心它的值。

Eric Desrosiers Adds:关于glHint(GL\_FOG\_HINT, hintval);的解释

gl\_dont\_care - 由OpenGL决定使用何种雾效 (对每个顶点还是每个像素执行雾效计算) 和一个未知的公式 (?)

gl\_nicest - 对每个像素执行雾效计算 (效果好)

gl\_fastest - 对每个顶点执行雾效计算 (更快, 但效果不如上面的好)

下一行glFogf(GL\_FOG\_START, 1.0f);设定雾效距屏幕多近开始。你可以根据你的需要随意改变这个值。下一行类似, glFogf(GL\_FOG\_END, 5.0f);告诉OpenGL程序雾效持续到距屏幕多远。

现在我们建立了绘制雾的代码, 我们将加入键盘指令在不同的雾效模式间循环。这段代码及其它的按键处理代码在程序的最后。

```
if (keys['G'] && !gp) // G键是否按下
{
    gp=TRUE;           // 是
    fogfilter+=1;      // 变换雾气模式
    if (fogfilter>2)   // 模式是否大于2
    {
        fogfilter=0;    // 置零
    }
    glFogi (GL_FOG_MODE, fogMode[fogfilter]); // 设置雾气模式
}
if (!keys['G']) // G键是否释放
{
    gp=FALSE;          // 是, 设置为释放
}
```

就是这样！大功告成！现在你的OpenGL程序有了雾效。



### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

### 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

### 感谢

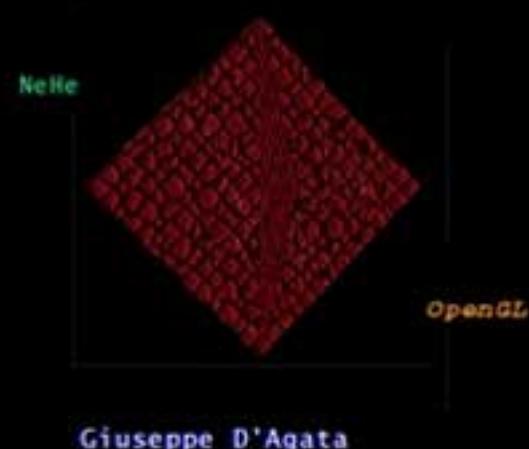
感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。





## 第17课



### 2D 图像文字:

在这一课中，你将学会如何使用四边形纹理贴图把文字显示在屏幕上。你将学会如何把256个不同的文字从一个256x256的纹理图像中分别提取出来，并为每一个文字创建一个显示列表，接着创建一个输出函数来创建任意你希望的文字。

本教程由NeHe和Giuseppe D'Agata提供。

我知道每个人都或许厌恶字体。目前为止我写的文字教程不仅能显示文字，还能显示3D文字，有纹理贴图的文字，以及处理变量。但是当你将你的作品移植到不支持位图或是轮廓字体的机器上会发生什么事呢？

由于Giuseppe D'Agata我们有了另一篇字体教程。你还会问什么？如果你记得在第一篇字体教程中我提到使用纹理在屏幕上绘制文字。通常当你使用纹理绘制文字时你会调用你最喜欢的图像处理程序，选择一种字体，然后输入你想显示的文字或段落。然后你保存位图并把它作为纹理读入到你的程序里。对一个需要很多文字或是文字在不停变化的程序来说这么做效率并不高。

本教程只使用有一个纹理来显示任意256个不同的字符。记住平均一个字符只有16个像素宽，大概16个像素高。如果你使用标准的256x256的纹理那么很明显你可以放入交叉的16个文字（即一个X），且最多16行16列。如果你需要一个更详细的解释：纹理是256个像素宽，一个字符是16个像素宽，256除以16得16:)

现在让我们来创建一个2D纹理字体demo ! 这课的程序基于第一课的代码。在程序的第一段，我们包括数学 ( math ) 和标准输入输出库 ( stdio )。我们需要数学库来使用正弦和余弦函数在屏幕上移动我们的文字，我们需要标准输入输出库来保证在我们制作纹理前要使用的位图实际存在。

我们将要加入一个变量base来指向我们的显示列表。我们还加入texture[2]来保存我们将要创建的两个纹理。Texture 1将是字体纹理，texture 2将是用来创建简单3D物体的凹凸纹理。

我们加入用来执行循环的变量loop。最后我们加入用来绕屏幕移动文字和旋转3D物体的cnt1和cnt2。

```
GLuint base;           // 绘制字体的显示列表的开始位置
GLuint texture[2];    // 保存字体纹理
GLuint loop;          // 通用循环变量

GLfloat cnt1;          // 字体移动计数器1
GLfloat cnt2;          // 字体移动计数器2
```

接下来是读取纹理代码。这跟前面纹理影射教程中的一模一样。

下面的代码同样对之前教程的代码改动很小。如果你不清楚下面每行的用途，回头复习一下。  
注意TextureImage[ ]将保存2个rgb图像记录。复查处理读取或存储纹理的纹理很重要。一个错误的数字可能导致内存溢出或崩溃！

```
int LoadGLTextures()                                // 载入位图(调用上面的代码)并转换成纹理
{
    int Status=FALSE;                               // 状态指示器
    AUX_RGBImageRec *TextureImage[2];              // 创建纹理的存储空间
```

下一行十分重要。如果你用别的数字替换2将发生严重问题。再查一次！这个数字应该与你在设置TextureImage[ ]时的数字相匹配。

我们将读取的纹理是font.bmp 和bumps.bmp。第二个纹理可用任何你想用的纹理替换。我不是特别有创造性，所以我使的纹理可能有些单调。

```
memset(TextureImage,0,sizeof(void *)*2); // 将指针设为 NULL

if ((TextureImage[0]=LoadBMP("Data/Font.bmp")) && // 载入字体图像
    (TextureImage[1]=LoadBMP("Data/Bumps.bmp"))) // 载入纹理图像
{
    Status=TRUE; // 将 Status 设为 TRUE
```

另一十分重要，要检查两遍的行。我无法开始告诉你我收到多少email问“为什么我只看到一个纹理，或为什么我的纹理是全白的！？！”通常问题都出在这行。如果你用1替换2，那么将只创建一个纹理，第二个纹理将显示为全白。如果你用3替换2，你的程序可能崩溃！

你应该只调用glGenTextures()一次。调用glGenTextures()后你应该创建你的所有纹理。我曾见过有人在每创建一个纹理前都加上一行glGenTextures()。这通常导致新建的纹理覆盖了你之前创建的。决定你需要创建多少个纹理是个好主意，调用glGenTextures()一次，然后创建所有的纹理。把glGenTextures()放进循环是不明智的，除非你有自己的理由。

```
glGenTextures(2, &texture[0]); // 创建纹理

for (loop=0; loop<2; loop++)
{
    // 生成所有纹理
    glBindTexture(GL_TEXTURE_2D, texture[loop]);
    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, TextureImage[loop]->sizeX, TextureImage[loop]->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
}
```

下面的几行代码检查我们读取的位图数据是否在内存里。如果是，释放内存。注意我们还要检查并释放rgb图像记录。如果我们使用了3个不同的图像来创建纹理，我们要检查并释放3个rgb图像记录。

```
for (loop=0; loop<2; loop++)
{
    if (TextureImage[loop]) // 纹理是否存在
    {
        if (TextureImage[loop]->data) // 纹理图像是否存在
        {
            free(TextureImage[loop]->data); // 释放纹理图像占用的内存
        }
        free(TextureImage[loop]); // 释放图像结构
    }
}
return Status; // 返回 Status
}
```

现在我们将创建字体。我将以同样的细节来解释这段代码。这并没那么复杂，但是有些数学要了解，我知道不是每个人都喜欢数学。

```
GLvoid BuildFont(GLvoid) // 创建我们的字符显示列表
{
```

下面两个变量将用来保存字体纹理中每个字的位置。cx将用来保存纹理中水平方向的位置，cy将用来保存纹理中竖直方向的位置。

```
float cx; // 字符的X坐标
float cy; // 字符的Y坐标
```

接着我们告诉OpenGL我们要建立256个显示列表。变量base将指向第一个显示列表的位置。第二个显示列表将是base+1，第三个是base+2，以此类推。

下面的第二行代码选择我们的字体纹理（texture[0]）。

```
base=glGenLists(256); // 创建256个显示列表
glBindTexture(GL_TEXTURE_2D, texture[0]); // 选择字符图象
```

现在我们开始循环。循环间创建所有的256个字符，每个存在它自己的显示列表里。

```
for (loop=0; loop<256; loop++)
{
    // 循环256个显示列表
```

下面的第一行或许看上去让人有点困惑。%符号表示loop除以16的余数。cx将我们通过字体纹理从左至右移动。你将注意到在后面的代码中我们用1减去cy从而从上到下而不是从下到上移动我们。%符号很难解释，但我将尝试去解释。

我们真正关心的是（loop%16）。/16只是将结果转化为纹理坐标。所以如果loop等于16，cx将等于16/16的余数也就是0。但cy将等于16/16也就是1。所以我们将下移一个字符的高度，且我们将不往右移。如果loop等于17，cx将等于17/16也就是1.0625。余数0.625也等于1/16。意味着我们将右移一个字符。cy将仍是1因为我们只关心小数点左边的数字。18/16将右移2个字符，但仍下移一个字符。如果loop是32，cx将再次等于0，因为32除以16没有余数，但cy将等于2。因为小数点左边的数字现在是2，将下移2个字符。这么讲清楚吗？

```
cx=float(loop%16)/16.0f; // 当前字符的X坐标
cy=float(loop/16)/16.0f; // 当前字符的Y坐标
```

Ok。现在我们通过从字体纹理中依据cx和cy的值选择一个单独的字符创建了2D字体。在下面的行里我们给base的值加上loop，若不这么做，每个字都将建在第一个显示列表里。我们当然不想要那样的事发生，所以通过给base加上loop，我们创建的每个字都被存在下个可用的显示列表里。

```
glNewList(base+loop,GL_COMPILE); //开始创建显示列表
```

现在我们已选择了我们要创建的显示列表，我们创建字符。这是通过绘制四边形，然后给他贴上字体纹理中的单个字符的纹理来完成的。

```
glBegin(GL_QUADS); // 使用四边形显示每一个字符
```

cx和cy应该保存一个从0.0到1.0的非常小的浮点数。如果cx和cy同时为0，下面第一行的代码将为：glTexCoord2f(0.0f,1-0.0f-0.0625f)。记得0.0625正是我们纹理的1/16，或者说是  
一个字符的宽/高。下面的纹理坐标将是我们纹理的左下角。

注意我们使用glVertex2i(x,y)而不是glVertex3f(x,y,z)。我们的字体是2D字体，所以我们不需要z值。因为我们使用的是正交投影，我们不需要移进屏幕。在一个正交投影平面绘图你所需的是指定x和y坐标。因为我们的屏幕是以像素形式从0到639（宽）从0到479（高），我们既不需用浮点数也不用负数:)

我们设置正交投影屏幕的方式是，(0,0)将是屏幕的左下角，(640,480)是屏幕的右上角。x轴上0是屏幕的左边界，639是右边界。y轴上0时下便捷，479是上便捷。基本上我们避免了负坐标。对那些不在乎透视，更愿意同像素而不是单元打交道的人来说更方便:)

```
glTexCoord2f(cx,1-cy-0.0625f); // 左下角的纹理坐标
glVertex2i(0,0); // 左下角的坐标
```

下一个纹理坐标现在是上个纹理坐标右边1/16（刚好一个字符宽）。所以这将是纹理的右下角。

```
glTexCoord2f(cx+0.0625f,1-cy-0.0625f); // 右下角的纹理坐标
glVertex2i(16,0); // 右下角的坐标
```

第三个纹理坐标在我们的字符的最右边，但上移了纹理的1/16（刚好一个字符高）。这将是一个单独字符的右上角。

```
glTexCoord2f(cx+0.0625f,1-cy); // 右上角的纹理坐标
glVertex2i(16,16); // 右上角的坐标
```

最后我们左移来设置字符左上角的最后一个纹理坐标。

```
glTexCoord2f(cx,1-cy); // 左上角的纹理坐标
glVertex2i(0,16); // 左上角的坐标
glEnd(); // 四边形字符绘制完成
```

最终，我们右移了10个像素，置于纹理的右边。如果我们不平移，文字将被绘制到各自的上面。由于我们的字体太窄，我们不想右移16个像素。如果那样的话，每个字之间将有很大间隔。只移动10个像素去除了间隔。

```
glTranslated(10,0,0); // 绘制完一个字符，向右平移16个单位
glEndList(); // 字符显示列表结束
} // 循环建立256个显示列表
}
```

下面这段代码与我们在其它字体教程中用来在程序退出前释放显示列表的相同。所有自base开始的256个显示列表都将被销毁（这样做很好！）。

GLvoid KillFont(GLvoid)

```
{
    glDeleteLists(base,256);                                // 从内存中删除256个显示列表
}
```

下一段代码将完成绘图。一切都几乎是新的，所以我将尽可能详细的解释每一行。一个小提示：很多都可加入这段代码，像是变量的支持，字体大小、间距的调整，和很多为恢复到我们决定打印前的状况所做的检查。

glPrint()有三个参数。第一个是屏幕上x轴上的位置（从左至右的位置），下一个y轴上的位置（从下到上...0是底部，越往上越大）。然后是字符串（我们想打印的文字），最后是一个叫做set的变量。如果你看过Giuseppe D'Agata制作的位图，你会注意到有两个不同的字符集。第一个字符集是普通的，第二个是斜体的。如果set为0，第一个字符集被选中。若set为1则选择第二个字符集。

```
GLvoid glPrint(GLint x, GLint y, char *string)           // 绘制字符
{
```

我们要做的第一件事是确保set的值非0即1。如果set大于1，我们将使它等于1。

```
if (set>1)                                              // 如果字符集大于1
{
    set=1;                                                 // 设置其为1
}
```

现在我们选择字体纹理。我们这么做是防止在我们决定往屏幕上输出东西时选择了不同的纹理。

```
glBindTexture(GL_TEXTURE_2D, texture[0]);                // 绑定为字体纹理
```

现在我们禁用深度测试。我这么做是因为混合的效果会更好。如果你不禁用深度测试，文字可能会被什么东西挡住，或得不到正确的混合效果。如果你不打算混合文字（那样文字周围的黑色区域就不会显示）你可以启用深度测试。

```
glDisable(GL_DEPTH_TEST); // 禁止深度测试
```

下面几行十分重要！我们选择投影矩阵。之后使用一个叫做glPushMatrix()的命令。glPushMatrix存储当前矩阵（投影）。有些像计算器的存储按钮。

```
glMatrixMode(GL_PROJECTION); // 选择投影矩阵  
glPushMatrix(); // 保存当前的投影矩阵
```

现在我们保存了投影矩阵，重置矩阵并设置正交投影屏幕。第一和第三个数字（0）表示屏幕的底边和左边。如果愿意我们可以将屏幕的左边设为-640，但如果不需要我们为什么要设负数呢。第二和第四个数字表示屏幕的上边和右边。将这些值设为你当前使用的分辨率是明智的做法。我们不需要用到深度，所以我们将z值设为-1与1。

```
glLoadIdentity(); // 重置投影矩阵  
glOrtho(0,640,0,480,-1,1); // 设置正投影的可视区域
```

现在我们选择模型视点矩阵，用glPushMatrix()保存当前设置。然后我们重置模型视点矩阵以便在正交投影视点下工作。

```
glMatrixMode(GL_MODELVIEW); // 选择模型变换矩阵  
glPushMatrix(); // 保存当前的模型变换矩阵  
glLoadIdentity(); // 重置模型变换矩阵
```

在保存了透视参数，设置了正交投影屏幕后，现在我们可以绘制文字了。我们从移动到绘制文字的位置开始。我们使用 glTranslated()而不是glTranslatef()因为我们处理的是像素，所以浮点值并不重要。毕竟，你不可能用半个像素：）

```
glTranslated(x,y,0); // 把字符原点移动到(x,y)位置
```

下面这行选择我们要使用的字符集。如果我们想使用第二个字符集，我们在当前的显示列表基数上加上128（128时我们256个字符的一半）。通过加上128，我们跳过了头128个字符。

```
glListBase(base-32+(128*set)); // 选择字符集
```

现在剩下的就是在屏幕上绘制文字了。我们同其它字体教程一样来完成这步。我们使用glCallLists()。strlen(string)是字符串的长度（我们想绘制多少字符），GL\_UNSIGNED\_BYTE意味着每个字符被表示为一个无符号字节（一个字节是一个从0到255的值）。最后，字符串保存我们想打印的文字。

```
glCallLists(strlen(string),GL_BYTE,string); // 把字符串写入到屏幕
```

现在我们所要做的是恢复透视视图。我们选择投影矩阵并用glPopMatrix()恢复我们先前用glPushMatrix()保存的设置。用相反的顺序恢复设置很重要。

```
glMatrixMode(GL_PROJECTION); // 选择投影矩阵  
glPopMatrix(); // 设置为保存的矩阵
```

现在我们选择模型视点矩阵，做相同的工作。我们使用glPopMatrix()恢复模型视点矩阵到我们设置正交投影显示之前。

```
glMatrixMode(GL_MODELVIEW); // 选择模型矩阵  
glPopMatrix(); // 设置为保存的矩阵
```

最后，我们启用深度测试。如果你没有在上面的代码中关闭深度测试，你不需要这行。

```
glEnable(GL_DEPTH_TEST); // 启用深度测试  
}
```

我们没有修改ReSizeGLScene()，所以我们直接跳到InitGL()。

```
int InitGL(GLvoid) // 此处开始对OpenGL进行所有设置  
{
```

我们跳到创建纹理的代码。如果由于某种原因创建纹理失败了，我们返回FALSE。这将让我们的程序知道发生了一个错误从而关闭程序。

```
if (!LoadGLTextures()) // 调用纹理载入子例程  
{  
    return FALSE; // 如果未能载入，返回FALSE  
}
```

如果没有错，我们跳到创建字体的代码。在创建字体时不会出什么错所以我们可以省略了错误检查。

```
BuildFont(); // 创建字符显示列表
```

现在我们做通常的GL设置。我们将背景色设为黑色，将深度清为1.0。我们选择一个深度测试模式和一个混合模式。我们启用平滑着色，最后启用2维纹理映射。

```

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);           // 黑色背景
glClearDepth(1.0);                            // 设置深度缓存
glDepthFunc(GL_LESS);                         // 所作深度测试的类型
glBlendFunc(GL_SRC_ALPHA,GL_ONE);             // 设置混合因子
glShadeModel(GL_SMOOTH);                     // 启用阴影平滑
glEnable(GL_TEXTURE_2D);                      // 启用纹理映射
return TRUE;                                  // 初始化成功
}

```

下面这段代码将完成绘图。我们先绘制3D物体最后绘制文字，这样文字将显示在3D物体上面，而不会被3D物体遮住。我之所以加入一个3D物体是为了演示透视投影和正交投影可同时使用。

```

int DrawGLScene(GLvoid)                      // 从这里开始进行所有的绘制
{
    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

```

我们选择bumps.bmp纹理来创建简单的小3D物体。为了看见3D物体，我们往屏幕内移动5个单位。我们绕z轴旋转45度。这将使我们的四边形顺时针旋转45度，让我们的四边形看起来更像钻石而不是矩形。

```

glBindTexture(GL_TEXTURE_2D, texture[1]);        // 设置为图像纹理
glTranslatef(0.0f,0.0f,-5.0f);                  // 移入屏幕5个单位
glRotatef(45.0f,0.0f,0.0f,1.0f);                // 沿Z轴旋转45度

```

在旋转45度后，我们让物体同时绕x轴和y轴旋转cnt1x30度。这使我们的物体象在一个点上旋转的钻石那样旋转。

```
glRotatef(cnt1*30.0f,1.0f,1.0f,0.0f); // 沿(1,1,0)轴旋转30度
```

我们关闭混合（我们希望3D物体看上去像实心的），设置颜色为亮白色。然后我们绘制一个单独的用了纹理映像的四边形。

```
glDisable(GL_BLEND); // 关闭混合
glColor3f(1.0f,1.0f,1.0f); // 设置颜色为白色
glBegin(GL_QUADS); // 绘制纹理四边形
    glTexCoord2d(0.0f,0.0f);
    glVertex2f(-1.0f, 1.0f);
    glTexCoord2d(1.0f,0.0f);
    glVertex2f( 1.0f, 1.0f);
    glTexCoord2d(1.0f,1.0f);
    glVertex2f( 1.0f,-1.0f);
    glTexCoord2d(0.0f,1.0f);
    glVertex2f(-1.0f,-1.0f);
glEnd();
```

在画完第一个四边形后，我们立即同时绕x轴和y轴旋转90度。然后我们画下一个四边形，。第二个四边形从第一个四边形的中间切过去，来形成一个好看的形象。

```
glRotatef(90.0f,1.0f,1.0f,0.0f); // 沿(1,1,0)轴旋转90度
glBegin(GL_QUADS); // 绘制第二个四边形，与第一个四边形垂直
    glTexCoord2d(0.0f,0.0f);
    glVertex2f(-1.0f, 1.0f);
    glTexCoord2d(1.0f,0.0f);
    glVertex2f( 1.0f, 1.0f);
    glTexCoord2d(1.0f,1.0f);
    glVertex2f( 1.0f,-1.0f);
    glTexCoord2d(0.0f,1.0f);
    glVertex2f(-1.0f,-1.0f);
```

```
glEnd();
```

在绘制完有纹理贴图的四边形后，我们开启混合并绘制文字。

```
glEnable(GL_BLEND); // 启用混合操作  
glLoadIdentity(); // 重置视口
```

我们使用同其它字体教程一样的生成很棒的颜色的代码。颜色会随着文字的移动而逐渐改变。

// 根据字体位置设置颜色

```
glColor3f(1.0f*float(cos(cnt1)),1.0f*float(sin(cnt2)),1.0f-0.5f*float(cos(cnt1+cnt2)));
```

我们来绘制文字。我们仍然使用glPrint()。第一个参数是x坐标，第二个是y坐标，第三个 ("NeHe") 是要绘制的文字，最后一个是使用的字符集 (0-普通，1-斜体)。

正如你猜的，我们使用SIN和COS连同计数器cnt1和cnt2来移动文字。如果你不清楚SIN和COS的作用，阅读之前的教程。

```
glPrint(int((280+250*cos(cnt1))),int(235+200*sin(cnt2)),"NeHe",0);
```

```
glColor3f(1.0f*float(sin(cnt2)),1.0f-0.5f*float(cos(cnt1+cnt2)),1.0f*float(cos(cnt1)));  
glPrint(int((280+230*cos(cnt2))),int(235+200*sin(cnt1)),"OpenGL",1);
```

我们将屏幕底部作者名字的颜色设为深蓝色和白色。然后用亮白色文字再次绘制他的名字。亮白色文字是有点偏蓝色的文字。这创造出一种附有阴影的样子。（如果混合没打开则没有这种效果）。

```
glColor3f(0.0f,0.0f,1.0f);
glPrint(int(240+200*cos((cnt2+cnt1)/5)),2,"Giuseppe D'Agata",0);

glColor3f(1.0f,1.0f,1.0f);
glPrint(int(242+200*cos((cnt2+cnt1)/5)),2,"Giuseppe D'Agata",0);
```

我们所做的最后一件事是以不同的速率递增我们的计数器。这使得文字移动，3D物体自转。

```
cnt1+=0.01f;           // 增加计数器值
cnt2+=0.0081f;         // 增加计数器值
return TRUE;           // 成功返回
}
```

KillGLWindow(), CreateGLWindow()和WndProc()的代码都没有更改，所以我们跳过它们。

如下所示，最后要做的是在KillGLWindow()的最后添加KillFont()。添加这行很重要，它在我们退出程序前将所有的清除干净。

```
KillFont();           // 删除字体
```

我认为现在我可以正式说我的网站已经把所有绘制文字的方法教给大家了{笑}。总之，我认为我的教程很不错。这课的代码可在任何能运行OpenGL的电脑上运行，它很容易使用，且这样绘制文字对系统的资源消耗很少。

我要感谢这篇教程的原作者Giuseppe D'Agata。我做了大量的修改，并将它转变为新式的代码，但要是没有他寄给我这份代码我是不会完成这篇教程的。他的代码有更多的选项，像是改变文字间距等等。但我用很cool的3D物体来弥补了{笑}。

我希望你们喜欢这篇教程。若有什么问题，给我或Giuseppe D'Agata发email。

Giuseppe D'Agata

Jeff Molofee (NeHe)

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

#### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

#### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知



识 , 我很欢迎你与我联系 , 但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹 , 一直以来默默的在精神上和生活中对我的支持 , 她甚至把买衣服的钱都用来给我买书了 , 她真的是我见过的最好的女孩 , 希望我能带给她幸福。

< 第16课

第18课 >



## 第18课



二次几何体:

利用二次几何体，你可以很容易的创建球，圆盘，圆柱和圆锥。

二次曲面是一种画复合对象的方法，这种方法通常并不需要很多的三角形。我们将要使用第七课的代码。我们将要增加7个变量以及修改纹理以增加一些变化：

```
bool sp; // 空格键是否按下  
  
int part1; // 圆盘的起始角度  
int part2; // 圆盘的结束角度  
int p1=0; // 增量1  
int p2=1; // 增量1  
GLUquadricObj *quadratic; // 二次几何体  
GLuint object=0; // 二次几何体标示符
```

好了我们现在开始搞InitGL()函数。我们打算增加3行代码用来初始化我们的二次曲面。这3行代码将在你使1号光源有效后增加，但是要在返回之前。第一行代码将初始化二次曲面并且创建一个指向改二次曲面的指针，如果改二次曲面不能被创建的话，那么该指针就是NULL。第二行代码将在二次曲面的表面创建平滑的法向量，这样当灯光照上去的时候将会好看些。另外一些可能的取值是：GLU\_NONE和GLU\_FLAT。最后我们使在二次曲面表面的纹理映射有效。

```
quadratic=gluNewQuadric();           // 创建二次几何体
gluQuadricNormals(quadratic, GLU_SMOOTH); // 使用平滑法线
gluQuadricTexture(quadratic, GL_TRUE); // 使用纹理
```

现在我决定在本课里保留立方体，这样你可以看到纹理是如何映射到二次曲面对象上的。而且我打算将绘制立方体的代码定义为一个单独的函数，这样我们在定义函数Draw()的时候它将会变的不那么凌乱。每个人都应该记住这些代码：

```
GLvoid glDrawCube()           // 绘制立方体
{
    glBegin(GL_QUADS);
    // 前面
    glNormal3f( 0.0f, 0.0f, 1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
    // 后面
    glNormal3f( 0.0f, 0.0f, -1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
    // 上面
    glNormal3f( 0.0f, 1.0f, 0.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
    // 下面
    glNormal3f( 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);
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
glNormal3f( 1.0f, 0.0f, 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);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
// 左面
glNormal3f(-1.0f, 0.0f, 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);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
glEnd();
}

```

接下来就是场景绘制函数了，在这里我只写一个简单的例子。并且当我绘制一个部分的盘子的时候，我将使用一个静态变量（一个局部的变量，该变量可以保留他的值不论你任何时候调用他）来表达一个非常酷的效果。为了清晰起见我将要重写DrawGLScene函数。

你们将会注意到当我讨论这些正在使用的参数时我忽略了当前函数的第一个参数（quadratic）。这个参数将被除立方体外的所有对象使用。所以当我讨论这些参数的时候我忽略了它。

```

int DrawGLScene(GLvoid)
{
    //...
// 这部分是新增加的
    switch(object)           // 绘制哪一种二次几何体
    {
        case 0:              // 绘制立方体
            glDrawCube();
            break;
    }
}

```

我们创建的第2个对象是一个圆柱体。参数1 ( 1.0F ) 是圆柱体的底面半径，参数2 ( 1.0F ) 是圆柱体的顶面半径，参数3 ( 3.0F ) 是圆柱体的高度。参数4 ( 32 ) 是纬线 ( 环绕Z轴有多少细分 )，参数5 ( 32 ) 是经线 ( 沿着Z轴有多少细分 )。细分越多该对象就越细致。我们可以用增加细分的方法来增加对象的多边形数。因此你可以牺牲速度换回质量 ( 以时间换质量 )，大多数的时候我们都可以很容易的找到一个合适的“度”。

```
case 1: // 绘制圆柱体
    glTranslatef(0.0f,0.0f,-1.5f);
    gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32);
    break;
```

对象3将会创建一个CD样子的盘子。参数1 ( 0.5F ) 是盘子的内圆半径，该参数可以为0，则表示在盘子中间没孔，内圆半径越大孔越大。参数2 ( 1.5F ) 表示外圆半径，这个参数必须比内圆半径大。参数3 ( 32 ) 是组成该盘子的切片的数量，这个数量可以想象成披萨饼中的切片的数量。切片越多，外圆边缘就越平滑。最后一个参数 ( 32 ) 是组成盘子的环的数量。环很像唱片上的轨迹，一环套一环。这些环从内圆半径细分到外圆半径。再说一次，细分越多，速度越慢。

```
case 2: // 绘制圆盘
    gluDisk(quadratic,0.5f,1.5f,32,32);
    break;
```

我们的第4个对象我知道你们为描述它耗尽精力。就是球。绘制球将会变的非常简单。参数1是球的半径。如果你无法理解半径/直径等等的话，可以理解成物体中心到物体外部的距离，在这里我们使用1.3F作为半径。接下来两个参数就是细分了，和圆柱体一样，参数2是纬线，参数3是经线。细分越多球看起来就越平滑，通常球需要多一些的细分以便他们看起来平滑。

```
case 3: // 绘制球
    gluSphere(quadratic,1.3f,32,32);
    break;
```

我们创建的第4个对象使用与我们曾经创建的圆柱体一样的命令来创建，如果你还记得的话，我们可以通过控制参数2和参数3来控制顶面半径和地面半径。因此我们可以使顶面半径为0来绘制一个圆锥体，顶面半径为0将会在顶面上创建一个点。因此在下面的代码中，我们使顶面半径等于0，这将会创建一个点，同时也创建了我们的圆锥。

```
case 4: // 绘制圆锥
    glTranslatef(0.0f,0.0f,-1.5f);
    gluCylinder(quadratic,1.0f,0.0f,3.0f,32,32);
    break;
```

我们的第6个对象将被gluPartialDisk函数创建。我们打算创建的这个对象使用了一些命令，这些命令在我们创建对象之前，你将会清楚的看到。但是命令gluPartialDisk拥有两个新的参数。第5个参数是我们想要绘制的部分盘子的开始角度，参数6是旋转角，也就是转过的角度。我们将要增加旋转角，这将引起盘子沿顺时针方向缓慢的被绘制在屏幕上。一旦旋转角达到360度我们将开始增加开始角度，这样盘子看起来就想是被逐渐的抹去一样。我们将重复这些过程。

```
case 5: // 绘制部分圆盘
    part1+=p1;
    part2+=p2;

    if(part1>359)
    {
        p1=0;
        part1=0;
        p2=1;
        part2=0;
    }
    if(part2>359)
    {
        p1=1;
        p2=0;
    }
    gluPartialDisk(quadratic,0.5f,1.5f,32,32,part1,part2-part1);
    break;
};
```

```
//...
```

```
}
```

In the KillGLWindow() section of code, we need to delete the quadratic to free up system resources. We do this with the command gluDeleteQuadratic.

```
GLvoid KillGLWindow(GLvoid)
{
    gluDeleteQuadric(quadratic);           // 删除二次几何体
```

在最后，我给出键盘输入代码。仅仅增加一些对剩余键的检查。

```
if (keys[' '] && !sp)      // 空格是否按下
{
    sp=TRUE;                // 是，则绘制下一种二次几何体
    object++;
    if(object>5)
        object=0;
}
if (!keys[' '])           // 空格是否释放
{
    sp=FALSE;              // 记录这个状态
}
```

这就是全部了。现在你可以在OpenGL中绘制二次曲面了。



### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

### 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。





## 第19课

粒子系统:

你是否希望创建爆炸，喷泉，流星之类的效果。这一课将告诉你如何创建一个简单的例子系统，并用它来创建一种喷射的效果。

欢迎来到第十九课.你已经学习了很多知识,并且现在想自己来实践.我将在这讲解一个新命令... 三角形带(我的理解就是画很多三角形来组合成我们要的形状),它非常容易使用,当画很多三角形的时候能加快你程序的运行速度.在本课中,我将会教你该如何做一个半复杂的微粒程序.一旦您了解微粒程序的原理后,在创建例如:火,烟,喷泉等效果将是很轻松的事情.我必须警告你!直到今天我从未写一个真正的粒子程序.我想写一个"出名"的复杂的粒子程序.我尝试过,但在我了解我不能控制所有点变疯狂之后我放弃了!!!你也许不相信我要告诉你的,但这个课程从头到尾都是我自己的想法.开始我没有一点想法,并且没有任何技术数据放在我的面前.我开始考虑粒子,突然我的脑袋装满想法(脑袋开启了??):给予每个粒子生命,任意变化颜色,速度,重力影响等等.来适应环境的变化,把每个粒子看成单一的从这个点运动到另一个点的颗粒.很快我完成了这个项目.我看时钟然后有个想法突然出现.四个小时过去了!我偶尔记得已经停止喝咖啡,眨眼,但是4个小时...?尽管这个程序我觉得很棒,并象我想象的那么严密的运行,但它不可能是最好的粒子引擎,这个我不关心,只要他运行好就可以.并且我能把它运行在我的项目中.如果你是那种想了解透彻的人,那么你要花费很多时间在网络上查找资料并弄明白它.在程序中有很多小的代码会看起来很模糊:)本课教程所用的部分代码来自于Lesson1.但有很多新的代码,因此我将重写一些发生代码变化的部分(使它更容易了解).

下面用到的代码来自于Lesson6,我将会增加5行新的代码在我们程序的前面.第一行"stdio.h"允许我们读文件中的数据.它和我们以前用在纹理映射当中是一样的.第二行定义了一

些我们要在屏幕上显示的粒子的数目.告诉程序MAX\_PARTICLES在这里的数值为1000.第三条行将不断分离的彩色的粒子栓牢在一起,并设置为默认情况.sp和rp用来确定空格键和返回键是否有按住.

```
#define MAX_PARTICLES 1000      // 定义最大的粒子数
bool rainbow=true;           // 是否为彩虹模式
bool sp;                     // 空格键是否被按下
bool rp;                     // 回车键是否被按下
```

下面四行是复杂的变量.变量slowdown控制粒子移动的快慢.数值愈高,移动越慢.数值越底,移动越快.如果数值降低,粒子将快速的移动!粒子的速度影响它们在荧屏中移动的距离.记住速度慢的粒子不会射很远的.变量xspeed和yspeed控制尾部的方向.xspeed将会增加粒子在x轴上速度.如果xspeed是正值粒子将会向右边移动多.如果xspeed负价值,粒子将会向左边移动多.那个值越高,就向那个方向移动比较多.yspeed工作相同的方法,但是在y轴上.因为有其它的因素影响粒子的运动,所以我要说"多".xspeed和yspeed有助于在我们想要的方向上移动粒子.最后是变量zoom,我们用该变量移入或移出我们的屏幕.在粒子引擎里,有时可看见更多的图象,而且当接近你时很酷

```
float slowdown=2.0f;          // 减速粒子
float xspeed;                 // X方向的速度
float yspeed;                 // Y方向的速度
float zoom=-40.0f;            // 沿Z轴缩放
```

我们定义了一个复杂的循环变量叫做Loop.我们用这变量预先定义粒子并在屏幕中画粒子.col用来给予粒子不同的颜色.delay用来控制在彩虹模式中圆的颜色变化.最后,我们设定一个存储空间(粒子纹理).我用纹理而不用点的重要原因是,点的速度慢,而且挺麻烦的.其次纹理很酷:)你用一个正方形的粒子,一张你脸的小图片,一张星星的图片等等.很好控制!

```
GLuint loop;                  // 循环变量
GLuint col;                   // 当前的颜色
GLuint delay;                 // 彩虹效果延迟
```

好!现在是有趣的东西.下段程序描述单一粒子结构,这是我们给予粒子的属性.我们用布尔型变量active开始,如果为true,我们的粒子为活跃的.如果为false则粒子为死的,此时我们就删除它.在程序中我没有使用活跃的,因为它很好的实现.变量life和fade来控制粒子显示多久以及显示时候的亮度.随着life数值的降低fade的数值也相应降低.这将导致一些粒子比其他粒子燃烧的时间长.

```
typedef struct // 创建粒子数据结构
{
    bool active; // 是否激活
    float life; // 粒子生命
    float fade; // 衰减速度
```

变量r,g和b用来表示粒子的红色强度,绿色强度和蓝色强度.当r的值变成1.0f时粒子将会很红,当三个变量全为1.0f时则粒子将变成白色.

```
float r; // 红色值
float g; // 绿色值
float b; // 蓝色值
```

变量x,y和z控制粒子在屏幕上显示的位置.x表示粒子在x轴上的位置.y表示y轴上的位置.z表示粒子z轴上的位置

```
float x; // X 位置
float y; // Y 位置
float z; // Z 位置
```

下面三个变量很重要.这三个变量控制粒子在每个轴上移动的快慢和方向.如果xi是负价粒子将会向左移动,正值将会向右移动.如果yi是负值粒子将会向下移动,正值将向上.最后,如果zi负值粒子将会向荧屏内部移动,正值将移向观察者.

```
float xi; // X 方向
```

```
float yi;           // Y 方向
float zi;           // Z 方向
```

最后,另外3个变量!每一个变量可被看成加速度.如果xg正值时,粒子将会被拉倒右边,负值将拉向左边.所以如果粒子向左移动(负的)而我们给它一个正的加速度,粒子速度将变慢.最后将向反方向移动(高中物理).yg拉下或拉上.zg拉进或拉出屏幕.

```
float xg;           // X 方向重力加速度
float yg;           // Y 方向重力加速度
float zg;           // Z 方向重力加速度
```

结构的名字为particles.

```
}
```

particles; // 粒子数据结构

下面我们创建一个数组叫particle.数组存储MAX\_PARTICLES个元素.也就是说我们创建1000(MAX\_PARTICLES)个粒子,存储空间为每个粒子提供相应的信息

```
particles particle[MAX_PARTICLES];           // 保存1000个粒子的数组
```

在颜色数组上我们减少一些代码来存储12中不同的颜色.对每一个颜色从0到11我们存储亮红,亮绿,和亮蓝.下面的颜色表里包含12个渐变颜色从红色到紫罗兰色

```
static GLfloat colors[12][3]=           // 彩虹颜色
{
    {1.0f,0.5f,0.5f},{1.0f,0.75f,0.5f},{1.0f,1.0f,0.5f},{0.75f,1.0f,0.5f},
    {0.5f,1.0f,0.5f},{0.5f,1.0f,0.75f},{0.5f,1.0f,1.0f},{0.5f,0.75f,1.0f},
```

```
{0.5f,0.5f,1.0f},{0.75f,0.5f,1.0f},{1.0f,0.5f,1.0f},{1.0f,0.5f,0.75f}
```

};

这段代码调用前面的代码载入位图，与前面的代码相同，只是位图的名称不同。载入一个名为Particle.bmp的位图

```
if (TextureImage[0]=LoadBMP("Data/Particle.bmp")) // 载入粒子纹理
```

窗口改变大小的代码和前面一样，不需要改变

我们使用光滑的阴影,清除背景为黑色,关闭深度测试,绑定并映射纹理.启用映射位图后我们选择粒子纹理。唯一的改变就是禁用深度测试和初始化粒子

```
glDisable(GL_DEPTH_TEST); //禁止深度测试
```

下面代码初始化每个粒子.我们从活跃的粒子开始.如果粒子不活跃,它在荧屏上将不出现,无论它有多少life.当我们使粒子活跃之后,我们给它life.我怀疑给粒子生命和颜色渐变的是否是最好的方法,但当它运行一次后,效果很好!life满值是1.0f.这也给粒子完整的光亮.

```
for (loop=0;loop<MAX_PARTICLES;loop++) //初始化所有的粒子
{
    particle[loop].active=true; // 使所有的粒子为激活状态
    particle[loop].life=1.0f; // 所有的粒子生命值为最大
```

我们通过给定的值来设定粒子退色快慢.每次粒子被拉的时候life随着fade而减小.结束的数值将是0~99中的任意一个,然后平分1000份来得到一个很小的浮点数.最后我们把结果加上0.003f来使fade速度值不为0

```
particle[loop].fade=float(rand()%100)/1000.0f+0.003f; // 随机生成衰减速率
```

既然粒子是活跃的,而且我们又给它生命,下面将给它颜色数值.一开始,我们就想每个粒子有不同的颜色.我怎么做才能使每个粒子与前面颜色箱里的颜色一一对应那?数学很简单,我们用loop变量乘以箱子中颜色的数目与粒子最大值(MAX\_PARTICLES)的余数.这样防止最后的颜色数值大于最大的颜色数值(12).举例:900\*(12/900)=12.1000\*(12/1000)=12,等等

```
particle[loop].r=colors[loop*(12/MAX_PARTICLES)][0]; // 粒子的红色颜色
particle[loop].g=colors[loop*(12/MAX_PARTICLES)][1]; // 粒子的绿色颜色
particle[loop].b=colors[loop*(12/MAX_PARTICLES)][2]; // 粒子的蓝色颜色
```

现在设定每个粒子移动的方向和速度.我们通过将结果乘于10.0f来创造开始时的爆炸效果.我们将会以任意一个正或负值结束.这个数值将以任意速度,任意方向移动粒子.

```
particle[loop].xi=float((rand()%50)-26.0f)*10.0f; // 随机生成X轴方向速度
particle[loop].yi=float((rand()%50)-25.0f)*10.0f; // 随机生成Y轴方向速度
particle[loop].zi=float((rand()%50)-25.0f)*10.0f; // 随机生成Z轴方向速度
```

最后,我们设定加速度的数值.不像一般的加速度仅仅把事物拉下,我们的加速度能拉出,拉下,拉左,拉右,拉前和拉后粒子.开始我们需要强大的向下加速度.为了达到这样的效果我们将xg设为0.0f.在x方向没有拉力.我们设yg为-0.8f来产生一个向下的拉力.如果值为正则拉向上.我们不希望粒子拉近或远离我们,所以将zg设为0.0f

```
particle[loop].xg=0.0f; // 设置X轴方向加速度为0
particle[loop].yg=-0.8f; // 设置Y轴方向加速度为-0.8
particle[loop].zg=0.0f; // 设置Z轴方向加速度为0
}
```

现在为有趣的部分.下面的部分是我们从哪里拉粒子,检查加速度等等.你要明白它是怎么实现的,因此仔细的看:)我们重置Modelview巨阵.在画粒子位置的时候用glVertex3f()命令来代替tranlations,这样在我们画粒子的时候不会改变modelview巨阵

```
int DrawGLScene(GLvoid) // 绘制粒子
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 以黑色背景清楚
    glLoadIdentity(); // 重置模型变换矩阵
```

我们开始创建一个循环loop.这个环将会更新每一个粒子.

```
for (loop=0;loop<MAX_PARTICLES;loop++) // 循环所有的粒子
{
```

首先我们做的事情是检查粒子是否活跃.如果不活跃,则不被更新.在这个程序中,它们始终活跃.但是在你自己的程序中,你可能想要使某粒子不活跃

```
if (particle[loop].active) // 如果粒子为激活的
{
```

下面三个变量是我们确定x,y和z位置的暂时变量.注意:在z的位置上我们加上zoom以便我们的场景在以前的基础上再移入zoom个位置.particle[loop].x告诉我们要画的x的位置.particle[loop].y告诉我们要画的y的位置.particle[loop].z告诉我们要画的z的位置

```
float x=particle[loop].x; // 返回X轴的位置
float y=particle[loop].y; // 返回Y轴的位置
float z=particle[loop].z+zoom; // 返回Z轴的位置
```

既然知道粒子位置,就能给粒子上色.particle[loop].r保存粒子的亮红,particle[loop].g保存粒子的亮绿,particle[loop].b保存粒子的亮蓝.注意我用alpha为粒子生命.当粒子要燃尽时,它会越来越透明直到它最后消失.这就是为什么粒子的生命不应该超过1.0f.如果你想粒子燃烧时间长,可降低fade减小的速度

```
// 设置粒子颜色
glColor4f(particle[loop].r,particle[loop].g,particle[loop].b,particle[loop].life);
```

我们有粒子的位置,并设置颜色了.所以现在我们来画我们的粒子.我们用一个三角形带代替一个四边形这样使程序运行快一点.很多3D card画三角形带比画四边形要快的多.有些3D card将四边形分成两个三角形,而有些不.所以我们按照我们自己的想法来,所以我们来画一个生动的三角形带

```
glBegin(GL_TRIANGLE_STRIP); // 绘制三角形带
```



从红宝书引述:三角形带就是画一连续的三角形(三个边的多角形)使用vertices V0,V1,V2,然后V2,V1,V3(注意顺序),然后V2,V3,V4等等.画三角形的顺序一样才能保证三角形带为相同的表面.要求方向是很重要的,例如:剔除,最少用三点来画当第一个三角形使用vertices0,1和2被画.如果你看图片你将会理解用顶点0,1和2构造第一个三角形(顶端右边,顶端左边,底部的右边).第二个三角形用点vertices2,1和3构造.再一次,如果你注意图片,点vertices2,1和3构造第二个三角形(底部右边,顶端左边,底部左边).注意:两个三角形画点顺序相同.我看到

很多的网站要求第二个三角形反方向画.这是不对的.OpenGL从新整理顶点来保证所有的三角形为同一方向!注意:你在屏幕上看见的三角形个数是你叙述的顶点的个数减2.在程序中在我们有4个顶点,所以我们看见二个三角形

```
glTexCoord2d(1,1); glVertex3f(x+0.5f,y+0.5f,z);
glTexCoord2d(0,1); glVertex3f(x-0.5f,y+0.5f,z);
glTexCoord2d(1,0); glVertex3f(x+0.5f,y-0.5f,z);
glTexCoord2d(0,0); glVertex3f(x-0.5f,y-0.5f,z);
```

最后我们告诉OpenGL我们画完三角形带

```
glEnd();
```

现在我们能移动粒子.下面公式可能看起来很奇怪,其实很简单.首先我们取得当前粒子的x位置.然后把x运动速度加上粒子被减速1000倍后的值.所以如果粒子在x轴(0)上屏幕中心的位置,运动值(xi)为x轴方向+10(移动我们为右),而slowdown等于1,我们移向右边以10/(1\*1000)或0.01f速度.如果增加slowdown值到2我们只移动0.005f.希望能帮助你了解slowdown如何工作.那也是为什么用10.0f乘开始值来叫象素移动快速,创造一个爆发效果.y和z轴用相同的公式来计算附近移动粒子

```
particle[loop].x+=particle[loop].xi/(slowdown*1000); // 更新X坐标的位置  
particle[loop].y+=particle[loop].yi/(slowdown*1000); // 更新Y坐标的位置  
particle[loop].z+=particle[loop].zi/(slowdown*1000); // 更新Z坐标的位置
```

在计算出下一步粒子移到那里,开始考虑重力和阻力.在下面的第一行,将阻力(xg)和移动速度(xi)相加.我们的移动速度是10和阻力是1.每时每刻粒子都在抵抗阻力.第二次画粒子时,阻力开始作用,移动速度将会从10掉到9.第三次画粒子时,阻力再一次作用,移动速度降低到8.如果粒子燃烧为超过10次重画,它将会最后结束,并向相反方向移动.因为移动速度会变成负值.阻力同样使用于y和z移动速度

```
particle[loop].xi+=particle[loop].xg; // 更新X轴方向速度大小  
particle[loop].yi+=particle[loop].yg; // 更新Y轴方向速度大小  
particle[loop].zi+=particle[loop].zg; // 更新Z轴方向速度大小
```

下行将粒子的生命减少.如果我们不这么做,粒子无法烧尽.我们用粒子当前的life减去当前的fade值.每粒子都有不同的fade值,因此他们全部将会以不同的速度烧尽

```
particle[loop].life-=particle[loop].fade; // 减少粒子的生命值
```

现在我们检查当生命为零的话粒子是否活着

```
if (particle[loop].life<0.0f) // 如果粒子生命值小于0
{
```

如果粒子是小时(烧尽),我们将会使它复原.我们给它全值生命和新的衰弱速度.

```
particle[loop].life=1.0f; // 产生一个新的粒子
particle[loop].fade=float(rand()%100)/1000.0f+0.003f; // 随机生成衰减速率
```

我们也重新设定粒子在屏幕中心放置.我们重新设定粒子的x,y和z位置为零

```
particle[loop].x=0.0f; // 新粒子出现在屏幕的中央
particle[loop].y=0.0f;
particle[loop].z=0.0f;
```

在粒子从新设置之后,将给它新的移动速度/方向.注意:我增加最大和最小值,粒子移动速度为从50到60的任意值,但是这次我们没将移动速度乘10.我们这次不想要一个爆发的效果,而要比较慢地移动粒子.也注意我把xspeed和x轴移动速度相加,y轴移动速度和yspeed相加.这个控制粒子的移动方向.

```
particle[loop].xi=xspeed+float((rand()%60)-32.0f); // 随机生成粒子速度
particle[loop].yi=yspeed+float((rand()%60)-30.0f);
particle[loop].zi=float((rand()%60)-30.0f);
```

最后我们分配粒子一种新的颜色.变量col保存一个数字从1到11(12种颜色),我们用这个变量去找红,绿,蓝亮度在颜色箱里面.下面第一行表示红色的强度,数值保存在colors[col][0].所以如果col是0,红色的亮度就是1.0f.绿色的和蓝色的值用相同的方法读取.如果你不了解为什么红色亮度为1.0f那col就为0.我将一点点的解释.看着程序的最前面.找到那行:static GLfloat colors[12][3].注意:12行3列.三个数字的第一行是红色强度.第二行是绿色强度而且第三行是蓝色强度.[0],[1]和[2]下面描述的1st,2nd和3rd就是我刚提及的.如果col等于0,我们要看第一个组.11是最最后一个组(第12种颜色).

```
particle[loop].r=colors[col][0];           // 设置粒子颜色  
particle[loop].g=colors[col][1];  
particle[loop].b=colors[col][2];  
}
```

下行描述加速度的数值是多少.通过小键盘8号键,我们增加yg(y 地心引力)值.这引起向上的力.如果这个程序在循环外面,那么我们必须生成另一个循环做相同的工作,因此我们最好放在这里

```
// 如果小键盘8被按住 , 增加Y轴方向的加速度  
if (keys[VK_NUMPAD8] && (particle[loop].yg<1.5f)) particle[loop].yg+=0.01f;
```

这行是产生相反的效果.通过2号键,减小yg值,引起向下的力

```
// 如果小键盘2被按住 , 减少Y轴方向的加速度  
if (keys[VK_NUMPAD2] && (particle[loop].yg>-1.5f)) particle[loop].yg-=0.01f;
```

现在更改向右的拉力.如果按下6号键时增加向右的拉力.

```
// 如果小键盘6被按住，增加X轴方向的加速度
if (keys[VK_NUMPAD6] && (particle[loop].xg<1.5f)) particle[loop].xg+=0.01f;
```

最后如果4号键被按下则增加向左的拉力.这些按键给了我们很酷的结果.举例来说:你可以用粒子造一条向上设的水流.通过增加向下的引力可以形成泉水

```
// 如果小键盘4被按住，减少X轴方向的加速度
if (keys[VK_NUMPAD4] && (particle[loop].xg>-1.5f)) particle[loop].xg-=0.01f;
```

我仅仅为乐趣增加了一些代码.我的兄弟产生很棒的效果:)通过按住tab键所有粒子都回到屏幕中心.所有的粒子在从新开始运动,再产生一个大的爆发.在粒子变弱之后,你最初的效果会再一次出现

```
if (keys[VK_TAB])
{
    particle[loop].x=0.0f;
    particle[loop].y=0.0f;
    particle[loop].z=0.0f;
    particle[loop].xi=float((rand()%50)-26.0f)*10.0f; // 随机生成速度
    particle[loop].yi=float((rand()%50)-25.0f)*10.0f;
    particle[loop].zi=float((rand()%50)-25.0f)*10.0f;
}
}
return TRUE; // 绘制完毕成功返回
}
```

代码KillGLWindow(),CreateGLWindow()和WndProc()中没有改变,所以我们直接跳到WinMain().我将重写代码

我喜欢简单的代码.在一行上不想包含很多东西,所以使代码象一个清洁工:)下面的代码检查"+是否被按下.如果它和slowdown一起实现则slowdown减少0.01f.粒子就可以较快速地移动.

```
if (keys[VK_ADD] && (slowdown>1.0f)) slowdown-=0.01f; // 按+号 , 加速粒子
```

下面的代码检查"-是否被按下.如果它和slowdown一起实现则slowdown增加0.01f.粒子就可以较慢速地移动.我实质的极限是4.0f,我不想它太慢的运动,你可以随你的要求改变最大最小值

```
if (keys[VK_SUBTRACT] && (slowdown<4.0f)) slowdown+=0.01f; // 按-号 , 减速粒
```

子

下面的代码检测Page Up是否被按下.如果是,则zoom增加.从而导致粒子靠近我们

```
if (keys[VK_PRIOR]) zoom+=0.1f; // 按Page Up键 , 让粒子靠近视点
```

下行代码检测Page Down是否别按下,如果是,则zoom减小.从而导师粒子离开我们

```
if (keys[VK_NEXT]) zoom-=0.1f; // 按Page Down , 让粒子远离视点
```

下面的代码检验enter键是否被按下.如果是,并且没有被一直按着,我们将让计算机把rp变为true,然后我们固定彩虹模式.如果彩虹模式为true,将其变成false.如果为false,将其变成true.最后一行检测enter是否被释放,如果释放rp为false并告诉计算机该键不被按下

```
if (keys[VK_RETURN] && !rp) // 按住回车键 , 切换彩虹模式
```

```

{
    rp=true;
    rainbow=!rainbow;
}
if (!keys[VK_RETURN]) rp=false;

```

下面程序有点乱.第一行检查space键是否被按下并没有一直按着.并检查彩虹模式是否开始运行,如果是,检查delay是否大于25.delay是我创建的显示彩虹效果的数值.如果你曾经改变颜色结构,粒子将显示不同颜色.通过创建一个delay,在颜色改变之前,一组粒子将是一种颜色.如果space按下,彩虹运行,delay值大于25则颜色改变

```

if ((keys[' '] && !sp) || (rainbow && (delay>25))) // 空格键 , 变换颜色
{

```

下面行是为了当space按下则彩虹关掉而设置的.如果我们不关掉彩虹模式,颜色会继续变化直到enter再被按下.也就是说人们按下space来代替enter是想叫粒子颜色自己变化

```
if (keys[' ']) rainbow=false;
```

如果space键被按下,或者彩虹模式已开始,并且delay大于25,我们叫计算机知道space键被按下通过叫sp为true.然后我们将delay设定回0以便它能在到25.最后我们增加col的值以便它通过颜色箱里面改变成另一个颜色.

```

sp=true;
delay=0;
col++;

```

如果颜色值大于11,我们把它重新设为零.如果不重新设定为零,程序将去找第13颜色.而我们只有12种颜色!寻找不存在的颜色将会导致程序瘫痪

```
if (col>11) col=0;  
}
```

最后如果space键不被按下,我们将sp设为false。

```
if (!keys[' ']) sp=false; // 如果释放空格键,记录这个状态
```

现在对粒子增加一些控制.还记得我们从开始定义的2变量么?一个xspeed,一个yspeed.在粒子燃尽之后,我们给它新的移动速度且把新的速度加入到xspeed和yspeed中.这样当粒子被创建时将影响粒子的速度.举例来说:粒子在x轴上的速度为5在y轴上的速度为0.当我们减少xspeed到-10,我们将以-10(xspeed)+5(最初的移动速度)的速度移动.这样我们将以5的速度向左移动.明白了么??无论如何,下面的代码检测UP是否被按下.如果它,yspeed将增加这将引起粒子向上运动.最大速度不超过200.速度太快就不好看了

```
// 按上增加粒子Y轴正方向的速度  
if (keys[VK_UP] && (yspeed<200)) yspeed+=1.0f;
```

这行检查Down键是否被按下,如果是,yspeed将减少.这将引起粒子向下运动.再一次,最大速度为200

```
// 按下减少粒子Y轴正方向的速度  
if (keys[VK_DOWN] && (yspeed>-200)) yspeed-=1.0f;
```

现在我们检查Right键是否被按下.如果是..xspeed将被增加.粒子将移到右边.最大速度为200

```
// 按右增加粒子X轴正方向的速度
```

```
if (keys[VK_RIGHT] && (xspeed<200)) xspeed+=1.0f;
```

最后我们检查Left键是否被按下.如果是...你猜....xspeed被减小,粒子开始向左移动.最大速度为200

```
// 按左减少粒子X轴正方向的速度  
if (keys[VK_LEFT] && (xspeed>-200)) xspeed-=1.0f;
```

最后我们要增加delay的数值.像我在前面所说,delay是控制彩色变化的

```
delay++; // 增加彩虹模式的颜色切换延迟
```

在课程中,我试着把所有细节都讲清楚,并且简单的了解粒子系统.这个粒子系统能在游戏中产生例如火,水,雪,爆炸,流行等效果.程序能简单的修改参数来实现新的效果(例:烟花效果)

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

#### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容,大多都来自共享的



资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

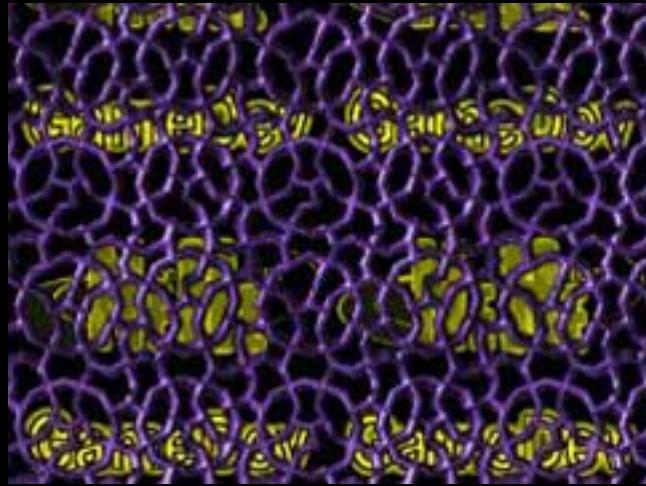
感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第18课

第20课 &gt;



## 第20课



蒙板:

到目前为止你已经学会如何使用alpha混合，把一个透明物体渲染到屏幕上上了，但有的使用它看起来并不是那么的复合你的心意。使用蒙板技术，将会按照你蒙板的位置精确的绘制。

欢迎来到第20课的教程，\*.bmp图像被给各种操作系统所支持，因为它简单，所以可以很轻松的作为纹理图片加载它。知道现在，我们在把图像加载到屏幕上时没有擦除背景色，因为这样简单高效。但是效果并不总是很好。

大部分情况下，把纹理混合到屏幕，纹理不是太少就是太多。当使用精灵时，我不希望背景从精灵的缝隙中透出光来；但在显示文字时，你希望文字的间隙可以显示背景色。

由于以上原因，我们需要使用“掩模”。使用“掩模”需要两个步骤，首先我们在场景上放置黑白相间的纹理，白色代表透明部分，黑色代表不透明部分。接着我们使用一种特殊的混合方式，只有在黑色部分上的纹理才会显示在场景中。

我只重写那些改变的地方，如果你做好了学习的准备，我们就上路吧。

在这个程序里，我们使用7个全局变量。变量masking为一个布尔值，标志是否使用“掩模”。变量mp标志键M是否按下，变量sp标志空格是否按下。

接着我们创建保存5个纹理标志的数组，loop为循环变量。变量roll使得纹理沿屏幕滚动。

```
bool masking=TRUE;           // 是否使用“掩模”
bool mp;                     // 键M是否按下
bool sp;                     // 空格是否按下
bool scene;                  // 绘制那一个场景

GLuint texture[5];          // 保存5个纹理标志
GLuint loop;                 // 循环变量

GLfloat roll;                // 滚动纹理
```

加载纹理代码基本没变，只是这里我们需要加载5个纹理

```
int LoadGLTextures()
{
    int Status=FALSE;
    AUX_RGBImageRec *TextureImage[5];           // 创建保存5个纹理的数据结构
    memset(TextureImage,0,sizeof(void *)*5);      // 初始化

    if ((TextureImage[0]=LoadBMP("Data/logo.bmp")) &&           // 加载纹理0
        (TextureImage[1]=LoadBMP("Data/mask1.bmp")) &&           // 加载掩模纹理1，作为“掩
模”使用
        (TextureImage[2]=LoadBMP("Data/image1.bmp")) &&           // 加载纹理1
        (TextureImage[3]=LoadBMP("Data/mask2.bmp")) &&           // 加载掩模纹理2，作为“掩
模”使用
        (TextureImage[4]=LoadBMP("Data/image2.bmp")))           // 加载纹理2
    {
        Status=TRUE;
        glGenTextures(5, &texture[0]);           // 创建5个纹理

        for (loop=0; loop<5; loop++)
        {
```

```

glBindTexture(GL_TEXTURE_2D, texture[loop]);
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, TextureImage[loop]->sizeX, TextureImage[loop]->sizeY,
0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
}
}
for (loop=0; loop<5; loop++)
{
if (TextureImage[loop])
{
if (TextureImage[loop]->data)
{
free(TextureImage[loop]->data);
}
free(TextureImage[loop]);
}
}
return Status;
}

```

改变窗口大小和初始化OpenGL的函数没有变化

现在到了最有趣的绘制部分了，我们从清楚背景色开始，接着把物体移入屏幕2个单位。

```

int DrawGLScene(GLvoid)
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
glTranslatef(0.0f,0.0f,-2.0f); // 物体移入屏幕2个单位

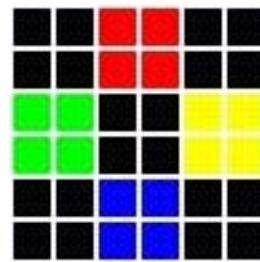
```

下面一行，我们选择'logo'纹理。我们将要通过四边形把纹理映射到屏幕，并按照顶点的顺序设置纹理坐标。

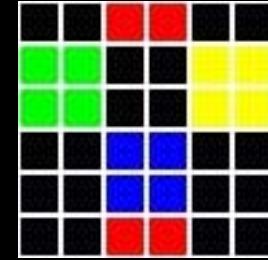
Jonathan Roy说OpenGL是一个基于顶点的图形系统，大部分你设置的参数是作为顶点的属性而记录的，纹理坐标就是这样一种属性。你只要简单的设置各个顶点的纹理坐标，OpenGL就自动帮你把多边形内部填充纹理，通过一个插值的过程。

向前面几课一样，我们假定四边形面对我们，并把纹理坐标(0,0)绑定到左下角，(1,0)绑定到右下角，(1,1)绑定到右上角。给定这些设置，你应该能猜到四边形中间对应的纹理坐标为(0.5,0.5)，但你自己并没有设置此处的纹理坐标！OpenGL为你做了计算。

在这一课里，我们通过设置纹理坐标达到一种滚动纹理的目的。纹理坐标是被归一化的，它的范围从0.0-1.0，值0被映射到纹理的一边，值1被映射到纹理的另一边。超过1的值，纹理可以按照不同的方式被映射，这里我们设置为它将回绕道另一边并重复纹理。例如如果使用这样的映射方式，纹理坐标(0.3,0.5)和(1.3,0.5)被映射到同一个纹理坐标。在这一课里，我们将尝试一种无缝填充的效果。



我们使用roll变量去设置纹理坐标，当它为0时，它把纹理的左下角映射到四边形的左下角。当它大于0时，把纹理的左上角映射到四边形的左下角，看起来的效果就是纹理沿四边形向上滚动。



```
glBindTexture(GL_TEXTURE_2D, texture[0]);           // 选择Logo纹理
glBegin(GL_QUADS);                                // 绘制纹理四边形
    glTexCoord2f(0.0f, -roll+0.0f); glVertex3f(-1.1f, -1.1f, 0.0f);
    glTexCoord2f(3.0f, -roll+0.0f); glVertex3f( 1.1f, -1.1f, 0.0f);
    glTexCoord2f(3.0f, -roll+3.0f); glVertex3f( 1.1f, 1.1f, 0.0f);
    glTexCoord2f(0.0f, -roll+3.0f); glVertex3f(-1.1f, 1.1f, 0.0f);
glEnd();
```

启用混合和禁用深度测试

```
glEnable(GL_BLEND); // 启用混合
glDisable(GL_DEPTH_TEST); // 禁用深度测试
```

接下来我们需要根据masking的值设置是否使用“掩模”，如果是，则需要设置相应的混合系数。

```
if (masking) // 是否启用“掩模”
{
```

如果启用了“掩模”，我们将要设置“掩模”的混合系数。一个“掩模”只是一幅绘制到屏幕的纹理图片，但只有黑色和白色。白色的部分代表透明，黑色的部分代表不透明。

下面这个混合系数使得，任何对应“掩模”黑色的部分会变为黑色，白色的部分会保持原来的颜色。

```
glBlendFunc(GL_DST_COLOR,GL_ZERO); // 使用黑白“掩模”混合屏幕颜色
}
```

现在我们检查绘制那一个层，如果为True绘制第二个层，否则绘制第一个层

```
if (scene)
{
```

为了不使它看起来显得非常大，我们把它移入屏幕一个单位，并把它按roll变量的值进行旋转（沿Z轴）。

```
glTranslatef(0.0f,0.0f,-1.0f);           // 移入屏幕一个单位
glRotatef(roll*360,0.0f,0.0f,1.0f);      // 沿Z轴旋转
```

接下我们检查masking的值来绘制我们的对象

```
if (masking)                         // “掩模”是否打开
{
```

如果“掩模打开”，我们会把掩模绘制到屏幕。当我们完成这个操作时，将会看到一个镂空的纹理出现在屏幕上。

```
glBindTexture(GL_TEXTURE_2D, texture[3]);    // 选择第二个“掩模”纹理
glBegin(GL_QUADS);                      // 开始绘制四边形
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.1f, 1.1f, 0.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.1f, 1.1f, 0.0f);
glEnd();
```

当我们把“掩模”绘制到屏幕上后，接着我们变换混合系数。这次我们告诉OpenGL把任何黑色部分对应的像素复制到屏幕，这样看起来纹理就像被镂空一样帖子屏幕上。

注意，我们在变换了混合模式后在选择的纹理。

如果我们没有使用“掩模”，我们的图像将与屏幕颜色混合。

```
glBlendFunc(GL_ONE, GL_ONE);           // 把纹理2复制到屏幕
glBindTexture(GL_TEXTURE_2D, texture[4]); // 选择第二个纹理
glBegin(GL_QUADS);                  // 绘制四边形
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f);
```

```

glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.1f, 1.1f, 0.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.1f, 1.1f, 0.0f);
glEnd();
}

```

### 绘制第一层图像

```

else
{

```

如果“掩模打开”，我们会把掩模绘制到屏幕。当我们完成这个操作时，将会看到一个镂空的纹理出现在屏幕上。

```

if (masking) // “掩模”是否打开
{

```

如果“掩模打开”，我们会把掩模绘制到屏幕。当我们完成这个操作时，将会看到一个镂空的纹理出现在屏幕上。

```

glBindTexture(GL_TEXTURE_2D, texture[1]); // 选择第一个“掩模”纹理
glBegin(GL_QUADS); // 开始绘制四边形
glTexCoord2f(roll+0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f);
glTexCoord2f(roll+4.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f);
glTexCoord2f(roll+4.0f, 4.0f); glVertex3f( 1.1f, 1.1f, 0.0f);
glTexCoord2f(roll+0.0f, 4.0f); glVertex3f(-1.1f, 1.1f, 0.0f);
glEnd();
}

```

当我们把“掩模”绘制到屏幕上后，接着我们变换混合系数。这次我们告诉OpenGL把任何黑色部分对应的像素复制到屏幕，这样看起来纹理就像被镂空一样帖子屏幕上。

注意，我们在变换了混合模式后在选择的纹理。

如果我们没有使用“掩模”，我们的图像将与屏幕颜色混合。

```
glBlendFunc(GL_ONE, GL_ONE);           // 把纹理1复制到屏幕
glBindTexture(GL_TEXTURE_2D, texture[2]); // 选择第一个纹理
glBegin(GL_QUADS);                   // 开始绘制四边形
    glTexCoord2f(roll+0.0f, 0.0f); glVertex3f(-1.1f, -1.1f, 0.0f);
    glTexCoord2f(roll+4.0f, 0.0f); glVertex3f( 1.1f, -1.1f, 0.0f);
    glTexCoord2f(roll+4.0f, 4.0f); glVertex3f( 1.1f, 1.1f, 0.0f);
    glTexCoord2f(roll+0.0f, 4.0f); glVertex3f(-1.1f, 1.1f, 0.0f);
glEnd();
}
```

接下来启用深度测试，禁用混合。

```
glEnable(GL_DEPTH_TEST);           // 启用深度测试
glDisable(GL_BLEND);             // 禁用混合
```

最后增加roll变量，如果大于1，把它的值减1。

```
roll+=0.002f;                      // 增加纹理滚动变量
if (roll>1.0f)                     // 大于1则减1
{
    roll-=1.0f;
}

return TRUE;                        // 成功返回
}
```

函数KillGLWindow(), CreateGLWindow() 和 WndProc() 没有改变。

接下来在wWinMain，我们添加键盘控制函数。我们检查空格是否按下，如果是则设置sp变量为TRUE，sp变量用来切换场景。

```
if (keys[' '] && !sp)           // 空格键是否被按下?  
{  
    sp=TRUE;  
    scene=!scene;           // 是则切换场景  
}
```

如果空格键释放，记录下来

```
if (!keys[' '])                // 如果空格键释放，记录下来  
{  
    sp=FALSE;  
}
```

我们检查M键是否按下，如果是则设置mp变量为TRUE，sp变量用来切换是否使用“掩模”

```
if (keys['M'] && !mp)          // M键是否被按下  
{  
    mp=TRUE;  
    masking=!masking;         // 是则切换“掩模”  
}
```

如果M键释放，记录下来

```
if (!keys['M']) // 如果M键释放，记录下来
{
    mp=FALSE;
}
```

Eric Desrosiers指出，你也可以在载入的时候测试\*.bmp图像中的每一个像素，如果你你想要透明的结果，你可以把颜色的alpha设置为0。对于其他的颜色，你可以把alpha设置为1。这个方法也能达到同样的效果，但需要一些额外的代码。

在这课里，我们给你演示了一个简单的例子，它能高效的绘制一部分纹理而不使用alpha值。

谢谢Rob Santa的想法和例子程序，我从没想到过这种方法。

我希望你喜欢这个教程，如果你在理解上有任何问题或找到了任何错误，请我知道，我想做最好的教程，你的反馈是非常重要的。

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

#### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容，大多都来自共享的



资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第19课

第21课 &gt;



## 第21课



线，反走样，计时，正投影和简单的声音：

这是我第一个大的教程，它将包括线，反走样，计时，正投影和简单的声音。希望这一课中的东西能让每个人感到高兴。

欢迎来到第21课，在这一课里，你将学会直线，反走样，正投影，计时，基本的音效和一个简单的游戏逻辑。希望这里的东西可以让你高兴。我花了两天的时间写代码，并用了两周的时间写这份HTML文件，希望你能享受我的劳动。

在这课的结尾你将获得一个叫"amidar"的游戏，你的任务是走完所有的直线。这个程序有了一个基本游戏的一切要素，关卡，生命值，声音和一个道具。

我们从第一课的程序来逐步完整这个程序，按照惯例，我们只介绍改动的部分。

```
#include <windows.h>
#include <stdio.h>
#include <stdarg.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>
```

```
HDC      hDC=NULL;
HGLRC    hRC=NULL;
HWND     hWnd=NULL;
HINSTANCE hInstance;
```

bool类型的变量，vline保存了组成我们游戏网格垂直方向上的121条线，上下水平各11条。hline保存了水平方向上的121条线，用ap来检查A键是否已经按下。

当网格被填满时，filled被设置为TRUE而反之则为FALSE。gameover这个变量的作用显而易见，当他的值为TRUE时，游戏结束。anti指出抗锯齿功能是否打开，当设置为TRUE时，该功能是打开着的。active 和 fullscreen 指出窗口是否被最小化以及游戏窗口是窗口模式还是全屏模式。

```
bool      keys[256];
bool      vline[11][10];           // 保存垂直方向的11根线条中，每根线条中的10段
是否被走过
bool      hline[10][11];          // 保存水平方向的11根线条中，每根线条中的10段
是否被走过
bool      ap;                   // A键是否已经按下
bool      filled;               // 网格是否被填满?
bool      gameover;              // 游戏是否结束?
bool      anti=TRUE;             // 是否启用反走样?
bool      active=TRUE;
bool      fullscreen=TRUE;
```

接着设置整型变量。loop1 和 loop2 被用来检查网格，查看是否有敌人攻击我们，以及在网格上给对象一个随机的位置。你将看到loop1 / loop2在后面的程序得到使用。delay 是一个计数器，我用他来减慢那些坏蛋的动作。当delay的值大于某一个阈值的时候，敌人才可以行动，此时delay将被重置。

adjust是一个非常特殊的变量，即使我们的程序拥有一个定时器，他也仅仅用来检查你的计算机是否运行地太快。如果是，则需要暂停一下以减慢运行速度。在我本地GeForce 显卡上，程序的运行平滑地简直变态，并且非常非常快。但是在我的PIII/450 + Voodoo 3500TV上测试的时候，我注意到程序运行地非常缓慢。我发现问题在于关于时间控制那部分代码只能够用来减慢游戏进行而并不能加速之。因此我引入了一个叫做adjust 的变量。它可以是0到5之间的任何值。游戏中的对象移动速度的不同依赖于这个变量的值。值越小，运动越平滑；而值越大，则运动速度越快。这是在比较慢的机器上运行这个程序最简单有效的解决方案了。但是请注意，不管对象移动的速度有多快，游戏的速度都不会比我期望的更快。我们推荐把adjust值设置为3，这样在大部分机器上都有比较满意

的效果。

我们把lives的值设置成5，这样我们的英雄一出场就拥有5条命。level是一个内部变量，用来指出当前游戏的难度。当然，这并不是你在屏幕上所看到的那个Level。变量level2开始的时候和Level拥有相同的值，但是随着你技能的提高，这个值也会增加。当你成功通过难度3之后，这个值也将在难度3上停止增加。level是一个用来表示游戏难度的内部变量，stage才是用来记录当前游戏关卡的变量。

```
int    loop1;           // 通用循环变量
int    loop2;           // 通用循环变量
int    delay;           // 敌人的暂停时间
int    adjust=3;         // 调整显示的速度
int    lives=5;          // 玩家的生命
int    level=1;          // 内部游戏的等级
int    level2=level;      // 显示的游戏的等级
int    stage=1;          // 游戏的关卡
```

接下来我们需要一个结构来记录游戏中的对象。fx和fy每次在网格上移动我们的英雄和敌人一些较小的象素，以创建一个平滑的动画效果。x和y则记录着对象处于网格的那个交点上。

上下左右各有11个点，因此x和y可以是0到10之间的任意值。这也是我们为什么需要fx和fy的原因。考虑如果我们只能够在上下和左右方向的11个点间移动的话，我们的英雄不得不

在各个点间跳跃前进。这样显然是不够平滑美观的。

最后一个变量spin用来使对象在Z轴上旋转。

```
struct    object           // 记录游戏中的对象
{
    int    fx, fy;          // 使移动变得平滑
    int    x, y;             // 当前游戏者的位置
    float   spin;            // 旋转方向
};
```

既然我们已经为我们的玩家，敌人，甚至是秘密武器。设置了结构体，那么同样的，为了表现刚刚创设的结构体的功能和特性，我们也可以为此设置新的结构体。

为我们的玩家创设结构体之下第一条直线。基本上我们将会为玩家提供fx , fy , x , y 和spin值几种不同的结构体。通过增加这些直线，仅需查看玩家的x值我们就很容易取得玩家的位置，同时我们也可以通过增加玩家的旋转度来改变玩家的spin值。

第二条直线略有不同。因为同一屏幕我们可以同时拥有至多15个敌人。我们需要为每个敌人创造上面所提到的可变量。我们通过设置一个有15个敌人的组来实现这个目标，如第一个敌人的位置被设定为敌人(0).x.第二个敌人的位置为(1), x等等

第三条直线使得为宝物创设结构体实现了可能。宝物是一个会时不在屏幕上出现的沙漏。我们需要通过沙漏来追踪x和y值。但是因为沙漏的位置是固定的所以我们不需要寻找最佳位置，而通过为程序后面的其他物品寻找好的可变量来实现（如fx和fy）

```
struct object player; // 玩家信息
struct object enemy[9]; // 最多9个敌人的信息
struct object hourglass; // 宝物信息
```

现在我们创建一个描述时间的结构，使用这个结构我们可以很轻松的跟踪时间变量。

接下来的第一步，就是创建一个64位的频率变量，它记录时间的频率。

resolution变量用来记录最小的时间间隔。

mm\_timer\_start和mm\_timer\_elapsed保存计时器开始时的时间和计时器开始后流失的时间。这两个变量只有当计算机不拥有performance counter时才启用。

变量performance\_timer用来标识计算机是否有performance counter

如果performance counter启用，最后两个变量用来保存计时器开始时的时间和计时器开始后流失的时间，它们比普通的根据精确。

```
struct // 保存时间信息的结构
{
    __int64    frequency; // 频率
    float      resolution; // 时间间隔
    unsigned long mm_timer_start; // 多媒体计时器的开始时间
    unsigned long mm_timer_elapsed; // 多媒体计时器的开始时间
```

```

bool      performance_timer;           // 使用Performance Timer?
__int64   performance_timer_start;    // Performance Timer计时器的开始时间
__int64   performance_timer_elapsed;  // Performance Timer计时器的开始时间
} timer;

```

下一行代码定义了速度表。如前所述，对象移动的速度依赖于值adjust，而以adjust为下标去检索速度表，就可以获得对象的移动速度。

```
int      steps[6]={ 1, 2, 4, 5, 10, 20 }; // 用来调整显示的速度
```

接下来我们将为纹理分配空间。纹理一共2张，一张是背景而另外一张是一张字体纹理。如本系列教程中的其他课程一样，base用来指出字符显示列表的基，同样的我们在最后声明了窗口过程WndProc()。

```

GLuint    texture[2];           // 字符纹理
GLuint    base;                // 字符显示列表的开始值

```

```
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);
```

接下来会是很有趣的工作。接下来的一段代码会初始化我们的计时器。代码会检查performance counter(非常精确的计数器)是否可用，如果不可用，则使用多媒体计时器。这段代码是可以移植的。

```

void TimerInit(void)           // 初始化我们的计时器
{
    memset(&timer, 0, sizeof(timer)); // 清空计时器结构

    // 检测Performance Counter是否可用，可用则创建
    if (!QueryPerformanceFrequency((LARGE_INTEGER *) &timer.frequency))
    {
        // 如果不可用
        timer.performance_timer = FALSE; // 设置Performance Timer为false
    }
}

```

```

timer.mm_timer_start = timeGetTime();           // 使用普通的计时器
timer.resolution     = 1.0f/1000.0f;             // 设置单位为毫秒
timer.frequency      = 1000;                    // 设置频率为1000
timer.mm_timer_elapsed = timer.mm_timer_start; // 设置流失的时间为当前的时间
}

```

如果performance counter 可用，则执行下面的代码：

```

else
{
    // 使用Performance Counter计时器
    QueryPerformanceCounter((LARGE_INTEGER *) &timer.performance_timer_start);
    timer.performance_timer = TRUE;           // 设置Performance Timer为TRUE
    // 计算计时的精确度
    timer.resolution = (float) (((double)1.0f)/((double)timer.frequency));
    // 设置流失的时间为当前的时间
    timer.performance_timer_elapsed = timer.performance_timer_start;
}
}

```

上面的代码设置了计时器，而下面的代码则读出计时器并返回已经经过的时间，以毫秒计。代码很简单，首先检查是否支持performance counter，若支持，则调用其相关函数；否则调用多媒体函数。

```

float TimerGetTime()                                // 返回经过的时间，以毫秒为单位
{
    __int64 time;                                  // 使用64位的整数
    if (timer.performance_timer)                   // 是否使用Performance Timer计时器？
    {
        QueryPerformanceCounter((LARGE_INTEGER *) &time); // 返回当前的时间
        // 返回时间差
        return ( (float) ( time - timer.performance_timer_start ) * timer.resolution)*1000.0f;
    }
    else

```

```

{
    // 使用普通的计时器，返回时间差
    return( (float) ( timeGetTime() - timer.mm_timer_start) * timer.resolution)*1000.0f;
}
}

```

在下面的代码里，我们把玩家重置在屏幕的左上角，而给敌人设置一个随机的位置。

```

void ResetObjects(void)                                // 重置玩家和敌人
{
    player.x=0;                                       // 把玩家置于左上角
    player.y=0;
    player.fx=0;
    player.fy=0;
}

```

接着我们给敌人一个随机的开始位置，敌人的数量等于难度乘上当前关卡号。记着，难度最大是3，而最多有3关。因此敌人最多有9个。

```

for (loop1=0; loop1<(stage*level); loop1++)          // 循环随即放置所有的敌人
{
    enemy[loop1].x=5+rand()%6;
    enemy[loop1].y=rand()%11;
    enemy[loop1].fx=enemy[loop1].x*60;
    enemy[loop1].fy=enemy[loop1].y*40;
}
}

```

并没有做任何改动，因此我将跳过它。在LoadGLTextures函数里我将载入那两个纹理 - 背景和字体。并且我会把这两副图都转化成纹理，这样我们就可以在游戏中使用它们。纹理创建好之后，象素数据就可以删除了。没有什么新东西，你可以阅读以前的课程以获得更多信息。

```
int LoadGLTextures()
{
    int Status=FALSE;
    AUX_RGBImageRec *TextureImage[2];
    memset(TextureImage,0,sizeof(void *)*2);
    if ((TextureImage[0]=LoadBMP("Data/Font.bmp")) &&           // 载入字体纹理
        (TextureImage[1]=LoadBMP("Data/Image.bmp")))           // 载入图像纹理
    {
        Status=TRUE;

        glGenTextures(2, &texture[0]);

        for (loop1=0; loop1<2; loop1++)
        {
            glBindTexture(GL_TEXTURE_2D, texture[loop1]);
            glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop1]->sizeX, TextureImage[loop1]-
>sizeY,
                         0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop1]->data);
            glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
            glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
        }

        for (loop1=0; loop1<2; loop1++)
        {
            if (TextureImage[loop1])
            {
                if (TextureImage[loop1]->data)
                {
                    free(TextureImage[loop1]->data);
                }
                free(TextureImage[loop1]);
            }
        }
    }
    return Status;
}
```

下面的代码建立了显示列表。对于字体的显示，我已经写过教程。在这里我把字体图象分成 $16 \times 16$ 个单元共256个字符。如果你有什么不明白，请参阅前面的教程

## GLvoid BuildFont(GLvoid)

```
{
    base=glGenLists(256);
    glBindTexture(GL_TEXTURE_2D, texture[0]);
    for (loop1=0; loop1<256; loop1++)
    {
        float cx=float(loop1%16)/16.0f;
        float cy=float(loop1/16)/16.0f;

        glNewList(base+loop1,GL_COMPILE);
        glBegin(GL_QUADS);
            glTexCoord2f(cx,1.0f-cy-0.0625f);
            glVertex2d(0,16);
            glTexCoord2f(cx+0.0625f,1.0f-cy-0.0625f);
            glVertex2i(16,16);
            glTexCoord2f(cx+0.0625f,1.0f-cy);
            glVertex2i(16,0);
            glTexCoord2f(cx,1.0f-cy);
            glVertex2i(0,0);
        glEnd();
        glTranslated(15,0,0);
    glEndList();
    }
}
```

当我们不再需要显示列表的时候，销毁它是一个好主意。在这里我仍然把代码加上了，虽然没有什么新东西。

## GLvoid KillFont(GLvoid)

```
{
    glDeleteLists(base,256);
}
```

函数没有做太多改变。唯一的改动是它可以打印变量了。我把代码列出这样你可以容易看到改动的地方。

请注意，在这里我激活了纹理并且重置了视图矩阵。如果set被置1的话，字体将被放大。我这样做是希望可以在屏幕上显示大一点的字符。在一切结束后，我会禁用纹理。

```
GLvoid glPrint(GLint x, GLint y, int set, const char *fmt, ...)  
{  
    char      text[256];  
    va_list   ap;  
  
    if (fmt == NULL)  
        return;  
  
    va_start(ap, fmt);  
    vsprintf(text, fmt, ap);  
    va_end(ap);  
  
    if (set>1)  
    {  
        set=1;  
    }  
    glEnable(GL_TEXTURE_2D);  
    glLoadIdentity();  
    glTranslated(x,y,0);  
    glListBase(base-32+(128*set));  
  
    if (set==0)  
    {  
        glScalef(1.5f,2.0f,1.0f);  
    }  
  
    glCallLists(strlen(text),GL_UNSIGNED_BYTE, text);  
    glDisable(GL_TEXTURE_2D);  
}
```

下面的代码基本没有变化，只是把透视投影变为了正投影

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)  
{  
    if (height==0)  
    {
```

```
height=1;  
}  
  
glViewport(0,0,width,height);  
  
glMatrixMode(GL_PROJECTION);  
glLoadIdentity();  
  
glOrtho(0.0f,width,height,0.0f,-1.0f,1.0f);  
  
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
}
```

初始化的代码和前面的代码相比没有什么改变

```
int InitGL(GLvoid)  
{  
    if (!LoadGLTextures())  
    {  
        return FALSE;  
    }  
  
    BuildFont();  
  
    glShadeModel(GL_SMOOTH);  
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);  
    glClearDepth(1.0f);  
    glHint(GL_LINE_SMOOTH_HINT, GL_NICEST);  
    glEnable(GL_BLEND);  
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);  
    return TRUE;  
}
```

下面是我们的绘制代码。

首先我们清空缓存，接着绑定字体的纹理，绘制游戏的提示字符串

```

int DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glBindTexture(GL_TEXTURE_2D, texture[0]);           // 选择字符纹理
    glColor3f(1.0f,0.5f,1.0f);
    glPrint(207,24,0,"GRID CRAZY");                  // 绘制游戏名称"GRID CRAZY"
    glColor3f(1.0f,1.0f,0.0f);
    glPrint(20,20,1,"Level:%2i",level2);             // 绘制当前的级别
    glPrint(20,40,1,"Stage:%2i",stage);              // 绘制当前级别的关卡
}

```

现在我们检测游戏是否结束，如果游戏结束绘制"Game over"并提示玩家按空格键重新开始

```

if (gameover)                                // 游戏是否结束?
{
    glColor3ub(rand()%255,rand()%255,rand()%255); // 随机选择一种颜色
    glPrint(472,20,1,"GAME OVER");               // 绘制 GAME OVER 字符串到屏幕
    glPrint(456,40,1,"PRESS SPACE");            // 提示玩家按空格键重新开始
}

```

在屏幕的右上角绘制玩家的剩余生命

```

for (loop1=0; loop1<lives-1; loop1++)          //循环绘制玩家的剩余生命
{
    glLoadIdentity();
    glTranslatef(490+(loop1*40.0f),40.0f,0.0f);   // 移动到屏幕右上角
    glRotatef(-player.spin,0.0f,0.0f,1.0f);        // 旋转绘制的生命图标
    glColor3f(0.0f,1.0f,0.0f);                    // 绘制玩家生命
    glBegin(GL_LINES);                           // 绘制玩家图标
    glVertex2d(-5,-5);
    glVertex2d( 5, 5);
    glVertex2d( 5,-5);
}

```

```

        glVertex2d(-5, 5);
    glEnd();
    glRotatef(-player.spin*0.5f,0.0f,0.0f,1.0f);
    glColor3f(0.0f,0.75f,0.0f);
    glBegin(GL_LINES);
        glVertex2d(-7, 0);
        glVertex2d( 7, 0);
        glVertex2d( 0,-7);
        glVertex2d( 0, 7);
    glEnd();
}

```

下面我们来绘制网格，我们设置变量filled为TRUE，这告诉程序填充网格。

接着我们把线的宽度设置为2，并把线的颜色设置为蓝色，接着我们检测线断是否被走过，如果走过我们设置颜色为白色。

```

filled=TRUE;                                // 在测试前，把填充变量设置为TRUE
glLineWidth(2.0f);                          // 设置线宽为2.0f
glDisable(GL_LINE_SMOOTH);                  // 禁用反走样
glLoadIdentity();                           // 循环11根线
for (loop1=0; loop1<11; loop1++)
{
    for (loop2=0; loop2<11; loop2++)          // 循环每根线的线段
    {
        glColor3f(0.0f,0.5f,1.0f);           // 设置线为蓝色
        if (hline[loop1][loop2])              // 是否走过？
        {
            glColor3f(1.0f,1.0f,1.0f);       // 是，设线为白色
        }
        if (loop1<10)                      // 绘制水平线
        {
            if (!hline[loop1][loop2])         // 如果当前线段没有走过，则不填充
            {
                filled=FALSE;
            }
            glBegin(GL_LINES);               // 绘制当前的线段
            glVertex2d(20+(loop1*60),70+(loop2*40));

```

```
    glVertex2d(80+(loop1*60),70+(loop2*40));  
    glEnd();  
}
```

下面的代码绘制垂直的线段

```
glColor3f(0.0f,0.5f,1.0f);           // 设置线为蓝色  
if (vline[loop1][loop2])           // 是否走过  
{  
    glColor3f(1.0f,1.0f,1.0f);       // 是，设线为白色  
}  
if (loop2<10)                     // 绘制垂直线  
{  
    if (!vline[loop1][loop2])        // 如果当前线段没有走过，则不填充  
    {  
        filled=FALSE;  
    }  
    glBegin(GL_LINES);             // 绘制当前的线段  
    glVertex2d(20+(loop1*60),70+(loop2*40));  
    glVertex2d(20+(loop1*60),110+(loop2*40));  
    glEnd();  
}
```

接下来我们检测长方形的四个边是否都被走过，如果被走过我们就绘制一个带纹理的四边形。

我们用下图来解释这个检测过程



如果对于垂直线vline的相邻两个边都被走过，并且水平线hline的相邻两个边也被走过，那么我们就可以绘制这个四边形了。我们可以使用循环检测每一个四边形，代码如下：

```

glEnable(GL_TEXTURE_2D);           // 使用纹理映射
glColor3f(1.0f,1.0f,1.0f);        // 设置为白色
glBindTexture(GL_TEXTURE_2D, texture[1]); // 绑定纹理
if ((loop1<10) && (loop2<10))    // 绘制走过的四边形
{
    // 这个四边形是否被走过?
    if (hline[loop1][loop2] && hline[loop1][loop2+1] && vline[loop1][loop2] && vline
    [loop1+1][loop2])
    {
        glBegin(GL_QUADS);          // 是，则绘制它
        glTexCoord2f(float(loop1/10.0f)+0.1f,1.0f-(float(loop2/10.0f)));
        glVertex2d(20+(loop1*60)+59,(70+loop2*40+1));
        glTexCoord2f(float(loop1/10.0f),1.0f-(float(loop2/10.0f)));
        glVertex2d(20+(loop1*60)+1,(70+loop2*40+1));
        glTexCoord2f(float(loop1/10.0f),1.0f-(float(loop2/10.0f)+0.1f));
        glVertex2d(20+(loop1*60)+1,(70+loop2*40)+39);
        glTexCoord2f(float(loop1/10.0f)+0.1f,1.0f-(float(loop2/10.0f)+0.1f));
        glVertex2d(20+(loop1*60)+59,(70+loop2*40)+39);
        glEnd();
    }
}
glDisable(GL_TEXTURE_2D);
}
glLineWidth(1.0f);

```

下面的代码用来设置是否启用直线反走样

```
if (anti) // 是否启用反走样?  
{  
    glEnable(GL_LINE_SMOOTH);  
}
```

为了使游戏变得简单些，我添加了一个时间停止器，当你吃掉它时，可以让追击的你的敌人停下来。

下面的代码用来绘制一个时间停止器。

```
if (hourglass.fx==1)  
{  
    glLoadIdentity();  
    glTranslatef(20.0f+(hourglass.x*60),70.0f+(hourglass.y*40),0.0f);  
    glRotatef(hourglass.spin,0.0f,0.0f,1.0f);  
    glColor3ub(rand()%255,rand()%255,rand()%255);  
  
    glBegin(GL_LINES);  
        glVertex2d(-5,-5);  
        glVertex2d( 5, 5);  
        glVertex2d( 5,-5);  
        glVertex2d(-5, 5);  
        glVertex2d( 5, 5);  
        glVertex2d(-5,-5);  
        glVertex2d( 5,-5);  
    glEnd();  
}
```

接下来绘制我们玩家

```
glLoadIdentity();
```

```
glTranslatef(player.fx+20.0f,player.fy+70.0f,0.0f);           // 设置玩家的位置  
glRotatef(player.spin,0.0f,0.0f,1.0f);                         // 旋转动画  
	glColor3f(0.0f,1.0f,0.0f);  
	glBegin(GL_LINES);  
		glVertex2d(-5,-5);  
		glVertex2d( 5, 5);  
		glVertex2d( 5,-5);  
		glVertex2d(-5, 5);  
	glEnd();
```

绘制玩家的显示效果，让它看起来更好看些（其实没用）

```
glRotatef(player.spin*0.5f,0.0f,0.0f,1.0f);  
	glColor3f(0.0f,0.75f,0.0f);  
	glBegin(GL_LINES);  
		glVertex2d(-7, 0);  
		glVertex2d( 7, 0);  
		glVertex2d( 0,-7);  
		glVertex2d( 0, 7);  
	glEnd();
```

接下来绘制追击玩家的敌人

```
for (loop1=0; loop1<(stage*level); loop1++)  
{  
	glLoadIdentity();  
	glTranslatef(enemy[loop1].fx+20.0f,enemy[loop1].fy+70.0f,0.0f);  
	glColor3f(1.0f,0.5f,0.5f);  
	glBegin(GL_LINES);  
		glVertex2d( 0,-7);  
		glVertex2d(-7, 0);  
		glVertex2d(-7, 0);  
		glVertex2d( 0, 7);  
		glVertex2d( 0, 7);  
		glVertex2d( 7, 0);
```

```
glVertex2d( 7, 0);
glVertex2d( 0,-7);
glEnd();
```

下面的代码绘制敌人的显示效果，让其更好看。

```
glRotatef(enemy[loop1].spin,0.0f,0.0f,1.0f);
	glColor3f(1.0f,0.0f,0.0f);
 glBegin(GL_LINES);
 glVertex2d(-7,-7);
 glVertex2d( 7, 7);
 glVertex2d(-7, 7);
 glVertex2d( 7,-7);
 glEnd();
}
return TRUE;
}
```

KillGLWindow函数基本没有变化，只在最后一行添加KillFont函数

```
GLvoid KillGLWindow(GLvoid)
{
    if (fullscreen)
    {
        ChangeDisplaySettings(NULL,0);
        ShowCursor(TRUE);
    }

    if (hRC)
    {
        if (!wglMakeCurrent(NULL,NULL))
        {
            MessageBox(NULL,"Release Of DC And RC Failed.", "SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
        }
    }
}
```

```
if (!wglDeleteContext(hRC))
{
    MessageBox(NULL,"Release Rendering Context Failed.", "SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
}
hRC=NULL;
}

if (hDC && !ReleaseDC(hWnd,hDC))
{
    MessageBox(NULL,"Release Device Context Failed.", "SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
    hDC=NULL;
}

if (hWnd && !DestroyWindow(hWnd))
{
    MessageBox(NULL,"Could Not Release hWnd.", "SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
    hWnd=NULL;
}

if (!UnregisterClass("OpenGL",hInstance))
{
    MessageBox(NULL,"Could Not Unregister Class.", "SHUTDOWN ERROR",MB_OK | MB_ICONINFORMATION);
    hInstance=NULL;
}

KillFont(); // 删除创建的字体
}
```

函数CreateGLWindow() and WndProc() 没有变化。

游戏控制在WinMain中完成的

```
int WINAPI WinMain( HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
```

```
LPSTR    lpCmdLine,
int      nCmdShow)
{
MSG   msg;
BOOL  done=FALSE;

if (MessageBox(NULL,"Would You Like To Run In Fullscreen Mode?", "Start FullScreen?", MB_YESNO|MB_ICONQUESTION)==IDNO)
{
    fullscreen=FALSE;
}
```

在创建完OpenGL窗口后，我们添加如下的代码，它用来创建玩家和敌人，并初始化时间计时器

```
if (!CreateGLWindow("NeHe's Line Tutorial",640,480,16,fullscreen))
{
    return 0;
}

ResetObjects();           // 重置玩家和敌人
TimerInit();              // 初始化时间计时器

while(!done)
{
    if (PeekMessage(&msg,NULL,0,0,PM_REMOVE))
    {
        if (msg.message==WM_QUIT)
        {
            done=TRUE;
        }
        else
        {
            TranslateMessage(&msg);
            DispatchMessage(&msg);
        }
    }
    else
    {
```

接下来取得当前的时间，并在速度快的机器上让其空循环，使得程序在所有的机器上都拥有同样的帧率

```
float start=TimerGetTime(); // 返回当前的时间

if ((active && !DrawGLScene()) || keys[VK_ESCAPE])
{
    done=TRUE;
}
else
{
    SwapBuffers(hDC);
}

while(TimerGetTime()<start+float(steps[adjust]*2.0f)) {}// 速度快的机器上让其空循环
```

下面的部分没有改变，按F1执行窗口和全屏的切换

```
if (keys[VK_F1])
{
    keys[VK_F1]=FALSE;
    KillGLWindow();
    fullscreen=!fullscreen;
    if (!CreateGLWindow("NeHe's Line Tutorial",640,480,16,fullscreen))
    {
        return 0;
    }
}
```

按A键切换是否启用反走样

```

if (keys['A'] && !ap)           // 如果'A' 键被按下 , 启用反走样
{
    ap=TRUE;
    anti=!anti;
}
if (!keys['A'])
{
    ap=FALSE;
}

```

如果游戏没有结束 , 执行游戏循环

```

if (!gameover && active)      // 如果游戏没有结束 , 则进行游戏循环
{
    for (loop1=0; loop1<(stage*level); loop1++) // 循环不同的难度等级
    {

```

根据玩家的位置 , 让敌人追击玩家

```

if ((enemy[loop1].x<player.x) && (enemy[loop1].fy==enemy[loop1].y*40))
{
    enemy[loop1].x++;
}

if ((enemy[loop1].x>player.x) && (enemy[loop1].fy==enemy[loop1].y*40))
{
    enemy[loop1].x--;
}

if ((enemy[loop1].y<player.y) && (enemy[loop1].fx==enemy[loop1].x*60))
{
    enemy[loop1].y++;
}

```

```

if ((enemy[loop1].y>player.y) && (enemy[loop1].fx==enemy[loop1].x*60))
{
    enemy[loop1].y--;
}

```

如果时间停止器的显示时间结束，而玩家又没有吃到，那么重置计时计算器。

```

if (delay>(3-level) && (hourglass.fx!=2))          // 如果没有吃到时间停止器
{
    delay=0;                                         // 重置时间停止器
    for (loop2=0; loop2<(stage*level); loop2++)   // 循环设置每个敌人的位置
    {

```

下面的代码调整每个敌人的位置，并绘制它们的显示效果

```

if (enemy[loop2].fx<enemy[loop2].x*60)
{
    enemy[loop2].fx+=steps[adjust];
    enemy[loop2].spin+=steps[adjust];
}
if (enemy[loop2].fx>enemy[loop2].x*60)
{
    enemy[loop2].fx-=steps[adjust];
    enemy[loop2].spin-=steps[adjust];
}
if (enemy[loop2].fy<enemy[loop2].y*40)
{
    enemy[loop2].fy+=steps[adjust];
    enemy[loop2].spin+=steps[adjust];
}
if (enemy[loop2].fy>enemy[loop2].y*40)
{
    enemy[loop2].fy-=steps[adjust];
    enemy[loop2].spin-=steps[adjust];
}

```

```
        }  
    }
```

如果敌人的位置和玩家的位置相遇，这玩家死亡，开始新的一局

```
// 敌人的位置和玩家的位置相遇?  
if ((enemy[loop1].fx==player.fx) && (enemy[loop1].fy==player.fy))  
{  
    lives--;           // 如果是，生命值减1  
  
    if (lives==0)      // 如果生命值为0，则游戏结束  
    {  
        gameover=TRUE;  
    }  
  
    ResetObjects();    // 重置所有的游戏变量  
    PlaySound("Data/Die.wav", NULL, SND_SYNC); // 播放死亡的音乐  
}  
}
```

使用上，下，左，右控制玩家的位置

```
if (keys[VK_RIGHT] && (player.x<10) && (player.fx==player.x*60) && (player.  
fy==player.y*40))  
{  
    hline[player.x][player.y]=TRUE;  
    player.x++;  
}  
if (keys[VK_LEFT] && (player.x>0) && (player.fx==player.x*60) && (player.fy==player.  
y*40))  
{  
    player.x--;  
    hline[player.x][player.y]=TRUE;  
}  
if (keys[VK_DOWN] && (player.y<10) && (player.fx==player.x*60) && (player.
```

```
fy==player.y*40))  
{  
    vline[player.x][player.y]=TRUE;  
    player.y++;  
}  
if (keys[VK_UP] && (player.y>0) && (player.fx==player.x*60) && (player.fy==player.  
y*40))  
{  
    player.y--;  
    vline[player.x][player.y]=TRUE;  
}
```

调整玩家的位置，让动画看起来跟自然

```
if (player.fx<player.x*60)  
{  
    player.fx+=steps[adjust];  
}  
if (player.fx>player.x*60)  
{  
    player.fx-=steps[adjust];  
}  
if (player.fy<player.y*40)  
{  
    player.fy+=steps[adjust];  
}  
if (player.fy>player.y*40)  
{  
    player.fy-=steps[adjust];  
}  
}
```

如果游戏结束，按空格开始新的一局游戏

```

else                                // 如果游戏结束
{
    if (keys[' '])                // 按下空格 ?
    {
        gameover=FALSE;           // 开始新的一局
        filled=TRUE;              // 重置所有的变量
        level=1;
        level2=1;
        stage=0;
        lives=5;
    }
}
}

```

如果顺利通过本关，播放通关音乐，并提高游戏难度，开始新的一局

```

if (filled)                          // 所有网格是否填满
{
    PlaySound("Data/Complete.wav", NULL, SND_SYNC); // 播放过关音乐
    stage++;                           // 增加游戏难度
    if (stage>3)                      // 如果当前的关卡大于3，则进入到下一个大的关卡?
    {
        stage=1;                      // 重置当前的关卡
        level++;                       // 增加大关卡的值
        level2++;
        if (level>3)
        {
            level=3;                  // 如果大关卡大于3，则不再增加
            lives++;                   // 完成一局给玩家奖励一条生命
            if (lives>5)              // 如果玩家有5条生命，则不再增加
            {
                lives=5;
            }
        }
    }
}

```

进入到下一关卡，重置所有的游戏变量

```

ResetObjects();

for (loop1=0; loop1<11; loop1++)
{
    for (loop2=0; loop2<11; loop2++)
    {
        if (loop1<10)
        {
            hline[loop1][loop2]=FALSE;
        }
        if (loop2<10)
        {
            vline[loop1][loop2]=FALSE;
        }
    }
}
}

```

如果玩家吃到时间停止器，记录这一信息

```

if ((player.fx==hourglass.x*60) && (player.fy==hourglass.y*40) && (hourglass.fx==1))
{
    // 播放一段声音
    PlaySound("Data/freeze.wav", NULL, SND_ASYNC | SND_LOOP);
    hourglass.fx=2;           // 设置fx为2，表示吃到时间停止器
    hourglass.fy=0;           // 设置fy为0
}

```

显示玩家的动画效果

```

player.spin+=0.5f*steps[adjust];          // 旋转动画
if (player.spin>360.0f)

```

```
{
    player.spin-=360;
}
```

### 显示时间停止器的动画

```
hourglass.spin-=0.25f*steps[adjust];           // 旋转动画
if (hourglass.spin<0.0f)
{
    hourglass.spin+=360.0f;
}
```

### 下面的代码计算何时出现一个时间停止计数器

下面的代码计算何时出现一个时间停止计数器

```
hourglass.fy+=steps[adjust];           // 增加fy的值，当他大于一定的时候，产生时间停止计数器
if ((hourglass.fx==0) && (hourglass.fy>6000/level))
{
    PlaySound("Data/hourglass.wav", NULL, SND_ASYNC);
    hourglass.x=rand()%10+1;
    hourglass.y=rand()%11;
    hourglass.fx=1;           //fx=1表示时间停止器出现
    hourglass.fy=0;
}
```

### 如果玩家没有拾取时间停止器，则过一段时间后，它自动消失

```
if ((hourglass.fx==1) && (hourglass.fy>6000/level))
{
    hourglass.fx=0;           // 消失后重置时间停止器
}
```

```
hourglass.fy=0;  
}
```

如果玩家吃到时间停止器，在时间停止停止阶段播放一段音乐，过一段时间停止播放音乐

```
if ((hourglass.fx==2) && (hourglass.fy>500+(500*level)))  
{  
    PlaySound(NULL, NULL, 0);          // 停止播放音乐  
    hourglass.fx=0;                  // 重置变量  
    hourglass.fy=0;  
}
```

增加敌人的延迟计数器的值，这个值用来更新敌人的运动

```
delay++;                      // 增加敌人的延迟计数器的值  
}  
}  
  
// 关闭  
KillGLWindow();                // 删除窗口  
return (msg.wParam);           // 退出程序  
}
```

我花了很多时间写这份教程，它开始于一个简单的直线教程，结束与一个小型的游戏。希望它能给你一些有用的信息，我知道你们中大部分喜欢那些基于“贴图”的游戏，但我觉得这些将教会你关于游戏更多的东西。如果你不同意我的看法，请让我知道，因为我想写最好的OpenGL教程。

请注意，这是一个很大的程序了。我尽量去注释每一行代码，我知道程序运行的一切细节，但把它表达出来又是另一回事。如果你有更好的表达能力，请告诉我如何更好的表达。我希望通过我们的努力，这份教程越来越好。谢谢

### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照

顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第20课

第22课 >



## 第22课



凹凸映射，多重纹理扩展：

这是一课高级教程，请确信你对基本知识已经非常了解了。这一课是基于第六课的代码的，它将建立一个非常酷的立体纹理效果。



这一课由Jens Schneider所写，它基本上是由第6课改写而来的，在这一课里，你将学习：

怎样控制多重纹理

怎样创建一个“假”的凹凸映射

怎样做一个标志，它看起来在你的场景上方

怎样使矩阵变化更有效率

基本的多通道渲染

因为上面提到的很多方面是高级渲染得内容，我们在讲述的时候会先说明理论，接着在分析代码。如果你已经熟悉了这些理论，你可以跳过他们，直接看代码。当你遇到什么问题的时候，不妨回过头来看看这些理论。

最后这份代码超过了1200行，大部分我们在前面的教程中遇到过了。我不会解释每一行代码，只在重要的地方做些提示，好了，让我们开始吧。

```
#include <string.h> // 字符串处理函数
```

MAX\_EMBOSS常量定义了突起的最大值

```
#define MAX_EMBOSS (GLfloat)0.01f // 定义了突起的最大值
```

好了 , 现在我们准备使用GL\_ARB\_multitexture这个扩展 , 它非常简单。

大部分图形卡不止一个纹理单元 , 为了利用这个功能 , 你必须检查GL\_ARB\_multitexture是否被支持 , 它可以使你同时把2个或多个不同的纹理映射到OpenGL图元上。开起来这个功能好像没有太大的作用 , 但当你使用多个纹理时 , 如果能同时把这些纹理值混合 , 而不使用费时的乘法运算 , 你将会得到很高的速度提高。

现在回到我们的代码 , \_\_ARB\_ENABLE用来设置是否使用ARB扩展。如果你想看你的OpenGL扩展 , 只要把#define EXT\_INFO前的注释去掉就行了。接着 , 我们在运行检查我们的扩展 , 以保证我们的程序可以在不同的系统上运行。所以我们需要一些内存保存扩展名的字符串 , 他们是下面两行。接着我们用一个变量multitextureSupported来标志当前系统是否能使用multitexture扩展 , 并用maxTexelUnits记录运行系统的纹理单元 , 这个值最少是1。

```
#define __ARB_ENABLE true // 使用它设置是否使用ARB扩展
// #define EXT_INFO // 把注释去掉,可以在启动时查看你的扩展
#define MAX_EXTENSION_SPACE 10240 // 保存扩展字符
#define MAX_EXTENSION_LENGTH 256 // 每个扩展字符串最大的长度
bool multitextureSupported=false; // 标志是否支持多重渲染
bool useMultitexture=true; // 如果支持,是否使用它
GLint maxTexelUnits=1; // 纹理处理单元的个数
```

下面的函数定义用来使用OpenGL的扩展函数，你可以把PFN-who-ever-reads-this看成是预先定义的函数类型，因为我们不清楚是否能得到这些函数的实体，所以先把他们都设置为NULL。glMultiTexCoordifARB函数是glTexCoord函数的扩展，它们的功能相似，其中i为纹理坐标的维数，f为数据的类型。最后两个函数用来激活纹理处理单元，可以使用户特定的纹理单元来绑定纹理。

顺便说一句，ARB是"Architectural Review Board"的缩写，用来定义这个组织提出的对OpenGL的扩展，并不强制OpenGL的实现必须包含这个功能，但他们希望这个功能得到广泛的支持。当前，只有multitexture被加入到ARB中，这从另一个方面支持multitexture的扩展将大大的提高渲染速度。

```
PFNGLMULTITEXCOORD1FARBPROC glMultiTexCoord1fARB = NULL;
PFNGLMULTITEXCOORD2FARBPROC glMultiTexCoord2fARB = NULL;
PFNGLMULTITEXCOORD3FARBPROC glMultiTexCoord3fARB = NULL;
PFNGLMULTITEXCOORD4FARBPROC glMultiTexCoord4fARB = NULL;
PFNGLACTIVETEXTUREARBPROC glActiveTextureARB = NULL;
PFNGLCLIENTACTIVETEXTUREARBPROC glClientActiveTextureARB= NULL;
```

下面我们来定义一些全局变量：

- filter 定义过滤器类型
- texture[3] 保存三个纹理
- bump[3] 保存三个凹凸纹理
- invbump[3] 保存三个反转了的凹凸纹理
- glLogo 保存标志
- multiLogo 保存多重纹理标志

```
GLuint filter=1; // 定义过滤器类型
GLuint texture[3]; // 保存三个纹理
GLuint bump[3]; // 保存三个凹凸纹理
GLuint invbump[3]; // 保存三个反转了的凹凸纹理
GLuint glLogo; // glLogo保存标志
GLuint multiLogo; // multiLogo保存多重纹理标志
GLfloat LightAmbient[] = { 0.2f, 0.2f, 0.2f}; // 环境光
GLfloat LightDiffuse[] = { 1.0f, 1.0f, 1.0f}; // 漫射光
GLfloat LightPosition[] = { 0.0f, 0.0f, 2.0f}; // 光源位置
GLfloat Gray[] = { 0.5f, 0.5f, 0.5f, 1.0f};
```

下面一块代码用来保存立方体的纹理和坐标，每5个数字描述一个顶点，包含2D的纹理坐标和3D的顶点坐标。

// 立方体的纹理和坐标

```
GLfloat data[] = {  
    // 前面  
    0.0f, 0.0f, -1.0f, -1.0f, +1.0f,  
    1.0f, 0.0f, +1.0f, -1.0f, +1.0f,  
    1.0f, 1.0f, +1.0f, +1.0f, +1.0f,  
    0.0f, 1.0f, -1.0f, +1.0f, +1.0f,  
    // 背面  
    1.0f, 0.0f, -1.0f, -1.0f, -1.0f,  
    1.0f, 1.0f, -1.0f, +1.0f, -1.0f,  
    0.0f, 1.0f, +1.0f, +1.0f, -1.0f,  
    0.0f, 0.0f, +1.0f, -1.0f, -1.0f,  
    // 上面  
    0.0f, 1.0f, -1.0f, +1.0f, -1.0f,  
    0.0f, 0.0f, -1.0f, +1.0f, +1.0f,  
    1.0f, 0.0f, +1.0f, +1.0f, +1.0f,  
    1.0f, 1.0f, +1.0f, +1.0f, -1.0f,  
    // 下面  
    1.0f, 1.0f, -1.0f, -1.0f, -1.0f,  
    0.0f, 1.0f, +1.0f, -1.0f, -1.0f,  
    0.0f, 0.0f, +1.0f, -1.0f, +1.0f,  
    1.0f, 0.0f, -1.0f, -1.0f, +1.0f,  
    // 右面  
    1.0f, 0.0f, +1.0f, -1.0f, -1.0f,  
    1.0f, 1.0f, +1.0f, +1.0f, -1.0f,  
    0.0f, 1.0f, +1.0f, +1.0f, +1.0f,  
    0.0f, 0.0f, +1.0f, -1.0f, +1.0f,  
    // 左面  
    0.0f, 0.0f, -1.0f, -1.0f, -1.0f,  
    1.0f, 0.0f, -1.0f, -1.0f, +1.0f,  
    1.0f, 1.0f, -1.0f, +1.0f, +1.0f,  
    0.0f, 1.0f, -1.0f, +1.0f, -1.0f  
};
```

下一部分代码，用来这运行时确定是否支持多重纹理的扩展。

首先，我们假定一个字符串包含了所有的扩展名，各个扩展名之间用'\n'分开。我们所要做的就是在其中查找是否有我们需要的扩展。如果成功找到则返回TRUE，否则返回FALSE。

```
bool isInString(char *string, const char *search) {
    int pos=0;
    int maxpos=strlen(search)-1;
    int len=strlen(string);
    char *other;
    for (int i=0; i<len; i++) {
        if ((i==0) || ((i>1) && string[i-1]=='\n')) {           // 新的扩展名开始与这里
            other=&string[i];
            pos=0;                                // 开始新的比较
            while (string[i]!='\n') {                // 比较整个扩展名
                if (string[i]==search[pos]) pos++;
                if ((pos>maxpos) && string[i+1]=='\n') return true; // 如果整个扩展名相同则成功返回
                i++;
            }
        }
    }
    return false;                                // 没找到
}
```

现在我们需要先取得扩展名字符串，并把它转换为以'\n'分割的字符串，接着调用以上定义的函数看看是否包含我们需要的扩展。如果定义了\_\_ARB\_ENABLE则使用多重纹理扩展，接下来我们检查是否支持GL\_EXT\_texture\_env\_combine扩展，这个扩展提供各个纹理单元复杂的交互方式，利用它可以完成复杂的混合方程。如果所有的扩展都被支持，我们首先取得纹理单元的个数，把它保存到变量maxTexelUnits中，接着通过函数wglGetProcAddress把各个函数定义连接到各自的实体上，这样在后面的程序中就可以使用这些函数了。

```
bool initMultitexture(void) {
    char *extensions;
```

```

extensions=strdup((char *) glGetString(GL_EXTENSIONS));           // 返回扩展名字符串
int len=strlen(extensions);
for (int i=0; i<len; i++)                                         // 使用'\n'分割各个扩展名
    if (extensions[i]==' ') extensions[i]='\n';

#ifndef EXT_INFO
    MessageBox(hWnd,extensions,"supported GL extensions",MB_OK | MB_ICONINFORMATION);
#endif

if (isInString(extensions,"GL_ARB_multitexture")                  // 是否支持多重纹理扩展 ?
    && __ARB_ENABLE                                                 // 是否使用多重纹理扩展 ?
    && isInString(extensions,"GL_EXT_texture_env_combine"))      // 是否支持纹理环境混合
{
    glGetIntegerv(GL_MAX_TEXTURE_UNITS_ARB,&maxTexelUnits);
    glMultiTexCoord1fARB = (PFNGLMULTITEXCOORD1FARBPROC) wglGetProcAddress
("glMultiTexCoord1fARB");
    glMultiTexCoord2fARB = (PFNGLMULTITEXCOORD2FARBPROC) wglGetProcAddress
("glMultiTexCoord2fARB");
    glMultiTexCoord3fARB = (PFNGLMULTITEXCOORD3FARBPROC) wglGetProcAddress
("glMultiTexCoord3fARB");
    glMultiTexCoord4fARB = (PFNGLMULTITEXCOORD4FARBPROC) wglGetProcAddress
("glMultiTexCoord4fARB");
    glActiveTextureARB = (PFNGLACTIVETEXTUREARBPROC) wglGetProcAddress
("glActiveTextureARB");
    glClientActiveTextureARB= (PFNGLCLIENTACTIVETEXTUREARBPROC) wglGetProcAddress
("glClientActiveTextureARB");

#ifndef EXT_INFO
    MessageBox(hWnd,"The GL_ARB_multitexture 扩展被使用.", "支持多重纹理",MB_OK |
MB_ICONINFORMATION);
#endif

    return true;
}
useMultitexture=false;                                              // 如果不支持多重纹理则返回false
return false;
}

```

初始化灯光

```
void initLights(void) {  
    glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);  
    glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);  
    glLightfv(GL_LIGHT1, GL_POSITION, LightPosition);  
    glEnable(GL_LIGHT1);  
}
```

下面我们加载许多纹理，这和前面的教程很像

```
int LoadGLTextures() {  
    // 载入*.bmp图像，并转换为纹理  
    bool status=true;  
    AUX_RGBImageRec *Image=NULL;  
    char *alpha=NULL;  
  
    // 加载基础纹理  
    if (Image=auxDIBImageLoad("Data/Base.bmp")) {  
        glGenTextures(3, texture);  
        // 创建3个纹理  
  
        // 创建使用临近过滤器过滤得纹理  
        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, GL_RGB8, Image->sizeX, Image->sizeY, 0, GL_RGB,  
        GL_UNSIGNED_BYTE, Image->data);  
  
        // 创建使用线形过滤器过滤得纹理  
        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, GL_RGB8, Image->sizeX, Image->sizeY, 0, GL_RGB,  
        GL_UNSIGNED_BYTE, Image->data);  
  
        // 创建使用线形Mipmap过滤器过滤得纹理  
        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);  
        gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB8, Image->sizeX, Image->sizeY, GL_RGB,
```

```

GL_UNSIGNED_BYTE, Image->data);
}
else status=false;

if (Image) { // 如果图像句柄存在，则释放图像回收资源
    if (Image->data) delete Image->data;
    delete Image;
    Image=NULL;
}

```

现在我们加载凹凸映射纹理。这个纹理必须使用50%的亮度（原因我们在后面介绍），我们使用glPixelTransferf函数完成这个功能。

另一个限制是我们不希望纹理重复贴图，只希望它粘贴一次，从纹理坐标(0,0)-(1,1)，所有大于它的纹理坐标都被映射到边缘，为了完成这个功能，我们使用glTexParameterf函数。

// 载入凹凸贴图

```

if (Image=auxDIBImageLoad("Data/Bump.bmp")) {
    glPixelTransferf(GL_RED_SCALE,0.5f); // 把颜色值变为原来的50%
    glPixelTransferf(GL_GREEN_SCALE,0.5f);
    glPixelTransferf(GL_BLUE_SCALE,0.5f);
    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_S,GL_CLAMP); //不使用重

```

复贴图

```

    glTexParameterf(GL_TEXTURE_2D,GL_TEXTURE_WRAP_T,GL_CLAMP);
    glGenTextures(3, bump); //创建凹凸贴图纹理

```

// 创建使用临近过滤器过滤得纹理

>...<

// 创建使用线形过滤器过滤得纹理

>...<

// 创建使用线形Mipmap过滤器过滤得纹理

>...<

反转凹凸贴图数据，创建三个反转的凹凸贴图纹理

```

for (int i=0; i<3*Image->sizeX*Image->sizeY; i++)           // 反转凹凸贴图数据
    Image->data[i]=255-Image->data[i];

glGenTextures(3, invbump);                                     // 创建三个反转了凹凸贴图

// 创建使用临近过滤器过滤得纹理
>...<

// 创建使用线形过滤器过滤得纹理
>...<

// 创建使用线形Mipmap过滤器过滤得纹理
>...<
}

else status=false;
if (Image) {                                                 // 如果图像存在，则删除
    if (Image->data) delete Image->data;
    delete Image;
    Image=NULL;
}

```

载入标志图像，图像是把颜色和alpha通道存为两张不同的bmp位图的，所以在处理的时候需要注意以下各个分量的位置。

```

if (Image=auxDIBImageLoad("Data/OpenGL_ALPHA.bmp")) {
    alpha=new char[4*Image->sizeX*Image->sizeY];
    for (int a=0; a<Image->sizeX*Image->sizeY; a++)
        alpha[4*a+3]=Image->data[a*3];
    if (!(Image=auxDIBImageLoad("Data/OpenGL.bmp"))) status=false;
    for (a=0; a<Image->sizeX*Image->sizeY; a++) {
        alpha[4*a]=Image->data[a*3];
        alpha[4*a+1]=Image->data[a*3+1];
        alpha[4*a+2]=Image->data[a*3+2];
    }
}

glGenTextures(1, &glLogo);                                    // 创建标志纹理

// 使用线形过滤器
glBindTexture(GL_TEXTURE_2D, glLogo);

```

```

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, GL_RGBA8, Image->sizeX, Image->sizeY, 0, GL_RGBA,
GL_UNSIGNED_BYTE, alpha);
    delete alpha;
}
else status=false;

if (Image) {                                // 如果图像存在，则删除
    if (Image->data) delete Image->data;
    delete Image;
    Image=NULL;
}

// 载入扩展标志纹理
if (Image=auxDIBImageLoad("Data/multi_on_alpha.bmp")) {
    alpha=new char[4*Image->sizeX*Image->sizeY];
    >...<
    glGenTextures(1, &multiLogo);
    >...<
    delete alpha;
}
else status=false;

if (Image) {                                // 如果图像存在，则删除
    if (Image->data) delete Image->data;
    delete Image;
    Image=NULL;
}
return status;
}

```

下面是窗口大小变化函数，没有任何改变。

接下来是绘制一个立方体的函数，它使用常规的方法绘制。

```

void doCube (void) {
    int i;
    glBegin(GL_QUADS);

```

```
// 前面
glNormal3f( 0.0f, 0.0f, +1.0f);
for (i=0; i<4; i++) {
    glTexCoord2f(data[5*i],data[5*i+1]);
    glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
}
// 后面
glNormal3f( 0.0f, 0.0f, -1.0f);
for (i=4; i<8; i++) {
    glTexCoord2f(data[5*i],data[5*i+1]);
    glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
}
// 上面
glNormal3f( 0.0f, 1.0f, 0.0f);
for (i=8; i<12; i++) {
    glTexCoord2f(data[5*i],data[5*i+1]);
    glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
}
// 下面
glNormal3f( 0.0f, -1.0f, 0.0f);
for (i=12; i<16; i++) {
    glTexCoord2f(data[5*i],data[5*i+1]);
    glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
}
// 右面
glNormal3f( 1.0f, 0.0f, 0.0f);
for (i=16; i<20; i++) {
    glTexCoord2f(data[5*i],data[5*i+1]);
    glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
}
// 左面
glNormal3f(-1.0f, 0.0f, 0.0f);
for (i=20; i<24; i++) {
    glTexCoord2f(data[5*i],data[5*i+1]);
    glVertex3f(data[5*i+2],data[5*i+3],data[5*i+4]);
}
}
glEnd();
}
```

现在到了OpenGL的初始化函数，它和前面的教程基本相同，只是添加了以下代码：

```
multitextureSupported=initMultitexture();  
initLights();
```

这里我们完成了95%的工作，下面我们来解释上面提到的原理。

### 开始理论讲解（凹凸映射）

如果你安装了Powerpoint-viewer,下面是一个讲解凹凸映射原理的PPT，你可以下载后慢慢研究：

**凹凸映射 作者Michael I. Gold, nVidia 公司**

如果你没有安装Powerpoint-viewer，我把它转换为Html格式，现讲解如下：

### 凹凸贴图

Michael I. Gold  
NVIDIA 公司

### 凹凸贴图

真实的凹凸贴图是逐像素计算的

光照计算是按每个象素点的法向量计算的  
巨大的计算量

更多的信息可以看: Blinn, J. Simulation of Wrinkled Surfaces. Computer Graphics. 12, 3 (August 1978), 286-292

凹凸贴图是在效果和精度之间的一个折中

只能对漫射光计算，不能使用反射光  
欺骗视觉的采样  
可能运行于当前的硬件上  
如果它看起来很好，就干吧

### 漫射光的计算

$$C = (L \cdot N) * Dl * Dm$$

L 顶点到灯之间的单位向量

N 顶点的单位法向量  
 DI 灯光的漫射光颜色  
 Dm 顶点材质的漫射属性  
 凸值 逐像素改变N值  
 凹凸映射 改变 ( $L^*N$ ) 的值

### 近似的漫射因子 $L^*N$

纹理图代表高度场

[0,1] 之间的高度代表凹凸方程  
 首先导出偏移度m  
 m 增加/减少基础的漫射因子Fd  
 $(Fd+m)$  在每一像素上近似等于  $(L^*N)$

偏移量m的导出

偏移量m的近似导出

查找(s,t)纹理的高度  $H_0$   
 查找( $s+ds, t+dt$ )纹理的高度  $H_1$   
 $M$  近似等于  $H_1 - H_0$

计算凹凸量



1) 原始凸起( $H_0$ ).



2) 原始的凸起( $H_0$ )向光源移动一小段距离形成第二个凸起( $H_1$ )



3) 用  $H_1$  凸起减去  $H_0$  凸起 ( $H_1 - H_0$ )

## 计算灯光亮度

### 计算片断的颜色Cf

$$C_f = (L^*N) \times D_l \times D_m$$

$$(L^*N) \sim (F_d + (H_1 - H_0))$$

$$C_t = D_m \times D_l$$

$$C_f = (F_d + (H_0 - H_1)) \times C_t$$

$F_d$ 等于顶点法线与灯光的向量的乘积

上面就是全部么？太简单了！

我们还没有完成所有的任务，还必须做以下内容：

创建一个纹理

计算纹理坐标偏移量 $ds, dt$

计算漫射因子 $F_d$

$ds, dt, F_d$ 都从 $N$ 和 $L$ 导出

现在我们开始做一些数学计算

## 创建纹理

### 保存纹理！

当前的多重纹理硬件只支持两个纹理

偏移值保存在alpha通道里

最大凸起值为 = 1.0

水平面值为 = 0.5

最小值为 = 0.0

颜色存储在RGB通道中

设置内部颜色格式为RGBA8 !!

## 计算纹理偏移量

把灯光方向向量变换到一个笛卡尔坐标系中

顶点法线为z轴

从法线和视口的“上”向量导出坐标系

顶点法线为z轴

又乘得到X轴

丢弃“上”向量，利用z, y轴导出x轴

创建3x3变换矩阵 $M_n$

变换灯光方向向量到这个坐标系中

计算纹理偏移量

使用法向坐标系中的向量作为偏移量

$$L' = M_n \times L$$

使用  $L'.x, L'.y$  作为  $ds, dt$

使用  $L'.z$  作为漫射因子!

如果灯光方向接近垂直，则  $L'.x, L'.y$  非常小

如果灯光方向接近水平，则  $L'.x, L'.y$  非常大

$L'.z$  小于零的含义?

灯光在法线的对面

在TNT上的实现

计算向量，纹理坐标

设置漫射因子

从纹理单元0取出表面颜色和H0值

从纹理单元1取出H1值

ARB\_multitexture 扩展

混合纹理扩展 (TBD)

混合0 alpha设置:

$$(1-T0a) + T1a - 0.5$$

T1a-T0a 映射到 [-1,1], 但硬件把它映射到 [0, 1]

T1a 为 H1 的值, T0a 为 H0 的值

0.5 平衡损失的掐除值

使用漫射光颜色调制 (相乘) 片断颜色 T0c

混合1 颜色设置 :

$$(T0c * C0a + T0c * Fda - 0.5) * 2$$

0.5 平衡损失的掐除值

乘以2加亮图像颜色

结束理论讲解 (凹凸映射)

虽然我们做了一些改动，使得这个程序的实现与TNT的实现不一样，但它能工作与各种不同的显卡上。在这里我们将学到两三件事，凹凸映射在大多数显卡上是一个多通道算法（在TNT系列，可以使用一个2纹理通道实现），现在你应该能想到多重纹理的好处了吧。我们将使用一个三通道非多重纹理的算法实现，这个算法可以被改写为使用一个2纹理通道实现的算法。

现在必须告诉你，我们将要做一些矩阵和向量的乘法，但那没有什么可担心的，所有的矩阵和向量都使用齐次坐标。

```
// 计算向量v=v*M (左乘)
void VMatMult(GLfloat *M, GLfloat *v) {
    GLfloat res[3];
    res[0]=M[ 0]*v[0]+M[ 1]*v[1]+M[ 2]*v[2]+M[ 3]*v[3];
    res[1]=M[ 4]*v[0]+M[ 5]*v[1]+M[ 6]*v[2]+M[ 7]*v[3];
    res[2]=M[ 8]*v[0]+M[ 9]*v[1]+M[10]*v[2]+M[11]*v[3];
    v[0]=res[0];
    v[1]=res[1];
    v[2]=res[2];
    v[3]=M[15];
}
```

## 开始理论讲解（凹凸映射）

开始，让我们看看它的算法

1. 所有的向量必须在物体空间或则世界空间中
2. 计算向量v，由灯的位置减去当前顶点的位置
3. 归一化向量v
4. 把向量v投影到切空间中
5. 安向量v在切空间中的投影偏移纹理坐标

这看起来不错，它基本上和Michael I. Gold介绍的方法差不多。但它有一个缺点，它只对xy平面进行投影，这对我们的应用还是不够的。

但这个实现在计算漫射光的方法和我们是一样的，我们不能存储漫射因子，所以我们不能使用Michael I. Gold介绍的方法，因为我们想让它在任何显卡上运行而不仅仅是TNT系列。为什么不光照计算留到最后呢？这在简单的几何体绘制上是可行的，如果你需要渲染几千个具有凹凸贴图的三角形，你会感到绘制的速度不够快，到那时你需要改变这种渲染过程，寻找其它的方法。

在我们的实现里，它看起来和上面的实现差不多，除了投影部分，我们将使用我们自己的近似。

我们使用模型坐标，这种设定可以使得灯光位置相对于物体不变。  
我们计算当前的顶点坐标

接着计算法线，并使它单位化

创建一个正投影矩阵，把灯光方向变为切空间

计算纹理坐标的偏移量， $ds = s$ 点乘 $v^*MAX\_EMBOSS$ ,  $dt = t$ 点乘 $v^*MAX\_EMBOSS$

在通道2中，把偏移量添加到纹理坐标

为什么更好:

更快

看起来好看

这个方法可以工作与各种表面

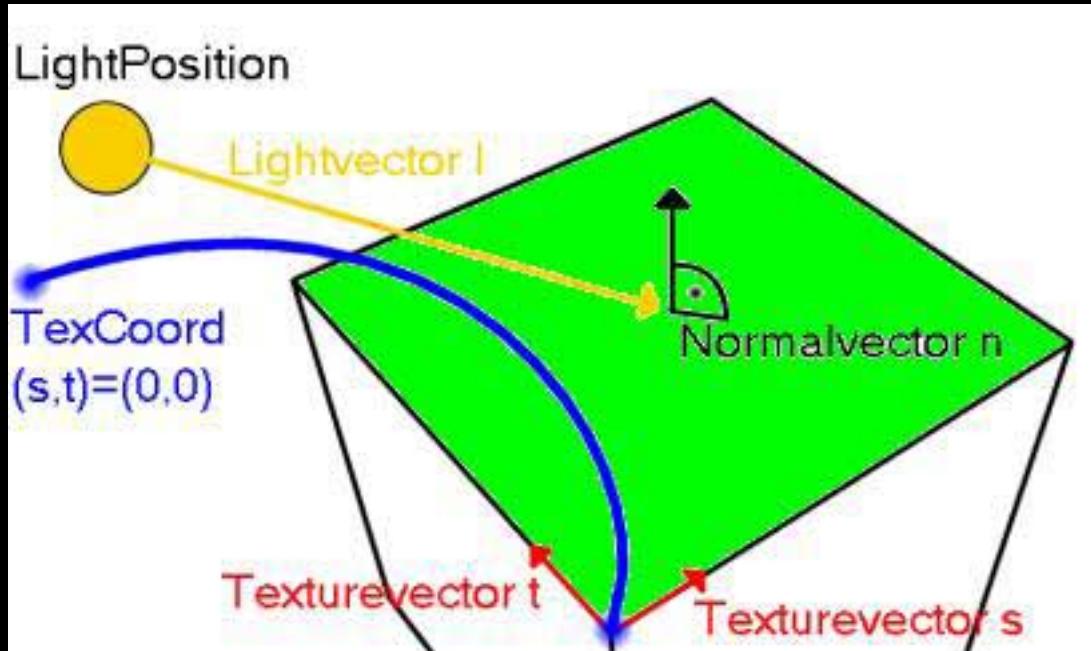
可以运行于各种显卡

最大化的兼容

缺陷:

并不是完全的物理模拟

残留一些人为的假相



这个示意图显示了我们坐标系统，你可以通过相减相邻的坐标来获得 $s$ ,  $t$ 向量，但必须保证他们构成右手系和归一化。

结束理论讲解（凹凸映射）

下面让我们看看如何生成偏移量，首先创建一个函数创建凹凸映射：

```

// 设置纹理偏移，都为单位长度
// n : 表面的法向量
// c : 当前的顶点纹理坐标，返回纹理坐标的偏移量
// l : 灯光的位置
// s : s方向
// t : t方向
void SetUpBumps(GLfloat *n, GLfloat *c, GLfloat *l, GLfloat *s, GLfloat *t) {
    GLfloat v[3];                                // 灯光方向
    GLfloat lenQ;                                // 灯光方向向量的长度，使用它来单位化
    // 计算灯光方向
    v[0]=l[0]-c[0];
    v[1]=l[1]-c[1];
    v[2]=l[2]-c[2];
    lenQ=(GLfloat)sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
    v[0]/=lenQ;
    v[1]/=lenQ;
    v[2]/=lenQ;
    // 把方向向量投影到s,t方向
    c[0]=(s[0]*v[0]+s[1]*v[1]+s[2]*v[2])*MAX_EMBOSS;
    c[1]=(t[0]*v[0]+t[1]*v[1]+t[2]*v[2])*MAX_EMBOSS;
}

```

那看起来复杂么，但为了理解这个效果理论是必须的。（我在写这篇教程的时候也学习了它）。

我在程序运行的时候，总喜欢在屏幕上显示标志，现在我们有了两个，使用doLogo函数创建它。

下面的函数显示两个标志：一个OpenGL的标志，一个多重纹理的标志，如果可以使用多重纹理，则标志使用alpha混合，并看起来半透明。为了让它在屏幕的边沿显示我们使用混合并禁用光照和深度测试。

```

void doLogo(void) {
    // 必须最后在调用这个函数，以公告板的形式显示两个标志
    glDepthFunc(GL_ALWAYS);
    glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);
    glEnable(GL_BLEND);
    glDisable(GL_LIGHTING);
}

```

```

glLoadIdentity();
glBindTexture(GL_TEXTURE_2D,glLogo);
glBegin(GL_QUADS);
    glTexCoord2f(0.0f,0.0f);    glVertex3f(0.23f, -0.4f,-1.0f);
    glTexCoord2f(1.0f,0.0f);    glVertex3f(0.53f, -0.4f,-1.0f);
    glTexCoord2f(1.0f,1.0f);    glVertex3f(0.53f, -0.25f,-1.0f);
    glTexCoord2f(0.0f,1.0f);    glVertex3f(0.23f, -0.25f,-1.0f);
glEnd();
if (useMultitexture) {
    glBindTexture(GL_TEXTURE_2D,multiLogo);
    glBegin(GL_QUADS);
        glTexCoord2f(0.0f,0.0f);    glVertex3f(-0.53f, -0.25f,-1.0f);
        glTexCoord2f(1.0f,0.0f);    glVertex3f(-0.33f, -0.25f,-1.0f);
        glTexCoord2f(1.0f,1.0f);    glVertex3f(-0.33f, -0.15f,-1.0f);
        glTexCoord2f(0.0f,1.0f);    glVertex3f(-0.53f, -0.15f,-1.0f);
    glEnd();
}
glDepthFunc(GL_LEQUAL);
}

```

现在到了绘制凹凸贴图的函数了，我们先来看看不使用多重映射的方法，它通过三个通道实现。在第一步，我们先取得模型变换矩阵的逆矩阵！

```

bool doMesh1TexelUnits(void) {
    GLfloat c[4]={0.0f,0.0f,0.0f,1.0f};           // 保存当前的顶点
    GLfloat n[4]={0.0f,0.0f,0.0f,1.0f};           // 保存法线
    GLfloat s[4]={0.0f,0.0f,0.0f,1.0f};           // s纹理坐标方向
    GLfloat t[4]={0.0f,0.0f,0.0f,1.0f};           // t纹理坐标方向
    GLfloat l[4];                                // 保存灯光方向
    GLfloat Minv[16];                            // 保存模型变换矩阵的逆
    int i;

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清空背景颜色和深度缓存

    // 创建模型变换矩阵的逆
    glLoadIdentity();
    glRotatef(-yrot,0.0f,1.0f,0.0f);
    glRotatef(-xrot,1.0f,0.0f,0.0f);
    glTranslatef(0.0f,0.0f,-z);

```

```
glGetFloatv(GL_MODELVIEW_MATRIX,Minv);
glLoadIdentity();
glTranslatef(0.0f,0.0f,z);
glRotatef(xrot,1.0f,0.0f,0.0f);
glRotatef(yrot,0.0f,1.0f,0.0f);

// 设置灯光的位置
l[0]=LightPosition[0];
l[1]=LightPosition[1];
l[2]=LightPosition[2];
l[3]=1.0f;
VMatMult(Minv,l);
```

### 通道1:

使用凹凸纹理  
禁止混合  
禁止光照  
使用无偏移的纹理坐标  
绘制几何体

这将渲染一个无凹凸贴图的几何体

```
glBindTexture(GL_TEXTURE_2D, bump[filter]);
glDisable(GL_BLEND);
glDisable(GL_LIGHTING);
doCube();
```

## 通道2:

- 使用反转的纹理凹凸贴图
- 设置混合因子为1 , 1
- 使用光照
- 使用偏移纹理坐标
- 绘制几何体

这将绘制一个具有凹凸贴图的几何体，但没有颜色

```
glBindTexture(GL_TEXTURE_2D,invbump[filter]);
glBlendFunc(GL_ONE,GL_ONE);
glDepthFunc(GL_LEQUAL);
 glEnable(GL_BLEND);

glBegin(GL_QUADS);
 // 前面
 n[0]=0.0f;
 n[1]=0.0f;
 n[2]=1.0f;
 s[0]=1.0f;
 s[1]=0.0f;
 s[2]=0.0f;
 t[0]=0.0f;
 t[1]=1.0f;
 t[2]=0.0f;
 for (i=0; i<4; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    // 设置纹理坐标为偏移后的纹理坐标
    glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
 }
 // 后面
 n[0]=0.0f;
 n[1]=0.0f;
 n[2]=-1.0f;
 s[0]=-1.0f;
```

```
s[1]=0.0f;
s[2]=0.0f;
t[0]=0.0f;
t[1]=1.0f;
t[2]=0.0f;
for (i=4; i<8; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
// 上面
n[0]=0.0f;
n[1]=1.0f;
n[2]=0.0f;
s[0]=1.0f;
s[1]=0.0f;
s[2]=0.0f;
t[0]=0.0f;
t[1]=0.0f;
t[2]=-1.0f;
for (i=8; i<12; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
// 下面
n[0]=0.0f;
n[1]=-1.0f;
n[2]=0.0f;
s[0]=-1.0f;
s[1]=0.0f;
s[2]=0.0f;
t[0]=0.0f;
t[1]=0.0f;
t[2]=-1.0f;
for (i=12; i<16; i++) {
    c[0]=data[5*i+2];
```

```
c[1]=data[5*i+3];
c[2]=data[5*i+4];
SetUpBumps(n,c,l,s,t);
glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
// 右面
n[0]=1.0f;
n[1]=0.0f;
n[2]=0.0f;
s[0]=0.0f;
s[1]=0.0f;
s[2]=-1.0f;
t[0]=0.0f;
t[1]=1.0f;
t[2]=0.0f;
for (i=16; i<20; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
// 左面
n[0]=-1.0f;
n[1]=0.0f;
n[2]=0.0f;
s[0]=0.0f;
s[1]=0.0f;
s[2]=1.0f;
t[0]=0.0f;
t[1]=1.0f;
t[2]=0.0f;
for (i=20; i<24; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glTexCoord2f(data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
glEnd();
```

### 通道3:

使用颜色纹理Use (colored) base-texture  
使用混合因子GL\_DST\_COLOR, GL\_SRC\_COLOR  
这个混合等于把颜色值乘以2  
使用光照  
绘制几何体

这个过程将结束立方体的渲染，因为我们在是否使用多重渲染之间切换，所以必须把纹理环境参数设为GL\_MODULATE，这是默认的值。

```
if (!emboss) {  
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);  
    glBindTexture(GL_TEXTURE_2D, texture[filter]);  
    glBlendFunc(GL_DST_COLOR, GL_SRC_COLOR);  
    glEnable(GL_LIGHTING);  
    doCube();  
}  
}
```

### 最后的通道:

更新几何体  
绘制标志

```
xrot+=xspeed;  
yrot+=yspeed;  
if (xrot>360.0f) xrot-=360.0f;  
if (xrot<0.0f) xrot+=360.0f;  
if (yrot>360.0f) yrot-=360.0f;  
if (yrot<0.0f) yrot+=360.0f;  
  
//绘制标志  
doLogo();  
return true; // 成功返回
```

}

这个函数将在多重纹理功能的支持下将两个通道中完成凹凸贴图的绘制，我们支持两个纹理单元，与一个纹理单元不同的是，我们给一个顶点设置两个纹理坐标。

```
bool doMesh2TexelUnits(void) {  
    GLfloat c[4]={0.0f,0.0f,0.0f,1.0f};           // 保存当前的顶点  
    GLfloat n[4]={0.0f,0.0f,0.0f,1.0f};           // 保存法线  
    GLfloat s[4]={0.0f,0.0f,0.0f,1.0f};           // s纹理坐标方向  
    GLfloat t[4]={0.0f,0.0f,0.0f,1.0f};           // t纹理坐标方向  
    GLfloat l[4];                                // 保存灯光方向  
    GLfloat Minv[16];                            // 保存模型变换矩阵的逆  
    int i;  
  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清空背景颜色和深度缓存  
  
    // 创建模型变换矩阵的逆  
    glLoadIdentity();  
    glRotatef(-yrot,0.0f,1.0f,0.0f);  
    glRotatef(-xrot,1.0f,0.0f,0.0f);  
    glTranslatef(0.0f,0.0f,-z);  
    glGetFloatv(GL_MODELVIEW_MATRIX,Minv);  
    glLoadIdentity();  
    glTranslatef(0.0f,0.0f,z);  
  
    glRotatef(xrot,1.0f,0.0f,0.0f);  
    glRotatef(yrot,0.0f,1.0f,0.0f);  
  
    // 设置灯光的位置  
    l[0]=LightPosition[0];  
    l[1]=LightPosition[1];  
    l[2]=LightPosition[2];  
    l[3]=1.0f;  
    VMatMult(Minv,l);
```

### 通道1:

无凹凸贴图  
无光照

### 设置纹理混合器0

使用凹凸纹理  
使用无偏移的纹理坐标  
使用替换方式粘贴纹理

### 设置纹理混合器1

偏移纹理坐标  
使用相加的纹理操作

这将绘制一个灰度的立方体

```
// 纹理单元 #0
glActiveTextureARB(GL_TEXTURE0_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, bump[filter]);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_REPLACE);

// 纹理单元 #1
glActiveTextureARB(GL_TEXTURE1_ARB);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, invbump[filter]);
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvf(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_ADD);

// 禁用混合和光照
glDisable(GL_BLEND);
glDisable(GL_LIGHTING);
```

现在按面一个一个的渲染立方体，和doMesh1TexelUnits函数中所作的操作差不多，只是用glMultiTexCoord2fARB替换glTexCoord2f，在这个函数中，你必须把纹理坐标发向不同的纹理处理单元，可用的参数值为GL\_TEXTUREi\_ARB0到GL\_TEXTUREi\_ARB31。

```
glBegin(GL_QUADS);
// 前面
n[0]=0.0f;
n[1]=0.0f;
n[2]=1.0f;
s[0]=1.0f;
s[1]=0.0f;
s[2]=0.0f;
t[0]=0.0f;
t[1]=1.0f;
t[2]=0.0f;
for (i=0; i<4; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
// 后面
n[0]=0.0f;
n[1]=0.0f;
n[2]=-1.0f;
s[0]=-1.0f;
s[1]=0.0f;
s[2]=0.0f;
t[0]=0.0f;
t[1]=1.0f;
t[2]=0.0f;
for (i=4; i<8; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
// 上面
```

```
n[0]=0.0f;
n[1]=1.0f;
n[2]=0.0f;
s[0]=1.0f;
s[1]=0.0f;
s[2]=0.0f;
t[0]=0.0f;
t[1]=0.0f;
t[2]=-1.0f;
for (i=8; i<12; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
// 下面
n[0]=0.0f;
n[1]=-1.0f;
n[2]=0.0f;
s[0]=-1.0f;
s[1]=0.0f;
s[2]=0.0f;
t[0]=0.0f;
t[1]=0.0f;
t[2]=-1.0f;
for (i=12; i<16; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
// 右面
n[0]=1.0f;
n[1]=0.0f;
n[2]=0.0f;
s[0]=0.0f;
s[1]=0.0f;
```

```
s[2]=-1.0f;
t[0]=0.0f;
t[1]=1.0f;
t[2]=0.0f;
for (i=16; i<20; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
// 左面
n[0]=-1.0f;
n[1]=0.0f;
n[2]=0.0f;
s[0]=0.0f;
s[1]=0.0f;
s[2]=1.0f;
t[0]=0.0f;
t[1]=1.0f;
t[2]=0.0f;
for (i=20; i<24; i++) {
    c[0]=data[5*i+2];
    c[1]=data[5*i+3];
    c[2]=data[5*i+4];
    SetUpBumps(n,c,l,s,t);
    glMultiTexCoord2fARB(GL_TEXTURE0_ARB,data[5*i], data[5*i+1]);
    glMultiTexCoord2fARB(GL_TEXTURE1_ARB,data[5*i]+c[0], data[5*i+1]+c[1]);
    glVertex3f(data[5*i+2], data[5*i+3], data[5*i+4]);
}
glEnd();
```

通道2 :

使用基本纹理  
使用光照  
使用普通的纹理混合操作

这将完成最后的凹凸贴图

```
glActiveTextureARB(GL_TEXTURE1_ARB);
glDisable(GL_TEXTURE_2D);
glActiveTextureARB(GL_TEXTURE0_ARB);
if (!emboss) {
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE);
    glBindTexture(GL_TEXTURE_2D, texture[filter]);
    glBlendFunc(GL_DST_COLOR, GL_SRC_COLOR);
    glEnable(GL_BLEND);
    glEnable(GL_LIGHTING);
    doCube();
}
```

### 最后的通道 :

更新几何体  
绘制标志

```
xrot+=xspeed;
yrot+=yspeed;
if (xrot>360.0f) xrot-=360.0f;
if (xrot<0.0f) xrot+=360.0f;
if (yrot>360.0f) yrot-=360.0f;
if (yrot<0.0f) yrot+=360.0f;

doLogo();
return true;
```

### 最后绘制一个无凹凸贴图的立方体 , 用来观察两者之间的效果

```
bool doMeshNoBumps(void) {
```

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glLoadIdentity();
glTranslatef(0.0f,0.0f,z);

glRotatef(xrot,1.0f,0.0f,0.0f);
glRotatef(yrot,0.0f,1.0f,0.0f);

if (useMultitexture) {
    glActiveTextureARB(GL_TEXTURE1_ARB);
    glDisable(GL_TEXTURE_2D);
    glActiveTextureARB(GL_TEXTURE0_ARB);
}

glDisable(GL_BLEND);
glBindTexture(GL_TEXTURE_2D,texture[filter]);
glBlendFunc(GL_DST_COLOR,GL_SRC_COLOR);
 glEnable(GL_LIGHTING);
doCube();

xrot+=xspeed;
yrot+=yspeed;
if (xrot>360.0f) xrot-=360.0f;
if (xrot<0.0f) xrot+=360.0f;
if (yrot>360.0f) yrot-=360.0f;
if (yrot<0.0f) yrot+=360.0f;

doLogo();
return true;
}
```

所有的绘制函数都已经完成，接下来只要在绘制函数中调用即可

```
bool DrawGLScene(GLvoid)
{
    if (bumps) {
        if (useMultitexture && maxTexelUnits>1)
            return doMesh2TexelUnits();
        else return doMesh1TexelUnits();      }
    else return doMeshNoBumps();
```

}

删除OpenGL窗口

GLvoid KillGLWindow(GLvoid)

创建OpenGL窗口

BOOL CreateGLWindow(char\* title, int width, int height, int bits, bool fullscreenflag)

Windows循环

```
LRESULT CALLBACK WndProc(HWND hWnd,
    UINT uMsg,
    WPARAM wParam,
    LPARAM lParam)
```

在Windows主函数中加入一些控制键 :

- E: 切换凹凸贴图模式中是否带有彩色纹理
- M: 切换多重纹理模式
- B: 切换是否使用凹凸贴图
- F: 切换纹理过滤器模式
- 方向键: 旋转立方体

```
if (keys['E'])
{
    keys['E']=false;
    emboss=!emboss;
}

if (keys['M'])
{
    keys['M']=false;
    useMultitexture=((!useMultitexture) && multitextureSupported);
}

if (keys['B'])
{
    keys['B']=false;
    bumps=!bumps;
}

if (keys['F'])
{
    keys['F']=false;
    filter++;
    filter%=3;
}

if (keys[VK_PRIOR])
{
    z-=0.02f;
}

if (keys[VK_NEXT])
{
    z+=0.02f;
}

if (keys[VK_UP])
{
    xspeed-=0.01f;
}

if (keys[VK_DOWN])
{
    xspeed+=0.01f;
```

```
        }  
  
        if (keys[VK_RIGHT])  
        {  
            yspeed+=0.01f;  
        }  
  
        if (keys[VK_LEFT])  
        {  
            yspeed-=0.01f;  
        }  
  
    }  
  
    glClear(GL_COLOR_BUFFER_BIT);  
    glLoadIdentity();  
    gluLookAt(0,0,0,0,1,0);  
    glTranslatef(0,0,y);  
    glRotatef(x,1,0,0);  
    glRotatef(y,0,1,0);  
    glScalef(0.5,0.5,0.5);  
    glBindTexture(GL_TEXTURE_2D, texture);  
    glBegin(GL_TRIANGLES);  
    glNormal3f(0,0,1);  
    glTexCoord2f(0,0);  
    glVertex3f(-0.5,-0.5,0);  
    glNormal3f(0,0,1);  
    glTexCoord2f(1,0);  
    glVertex3f(0.5,-0.5,0);  
    glNormal3f(0,0,1);  
    glTexCoord2f(1,1);  
    glVertex3f(0.5,0.5,0);  
    glEnd();  
    glutSwapBuffers();  
}  
  
int main(int argc, char **argv)  
{  
    glutInit(&argc, argv);  
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);  
    glutInitWindowSize(640, 480);  
    glutCreateWindow("Convex Hull");  
    glutDisplayFunc(display);  
    glutIdleFunc(idle);  
    glutMainLoop();  
}
```

现在你应该可以熟练的使用凹凸贴图了，如果你想让你的具有凹凸贴图的程序跑起来更快，你应该注意以下几点：

你不应该使用256x256的纹理，这会让处理变得缓慢。

一个具有凹凸贴图的立方体是不常见的，这和你的视角有关，因为三角面过于大了，如果要获得很好的视觉效果，你需要很大的纹理贴图，这必然会降低渲染速度。你可以把模型创建为一些小的三角形，从而使用小的纹理，来获得好的效果。

你应该先创建颜色纹理，接着把它转换为具有深度的凹凸纹理

凹凸纹理应该锐化，这可以取得更好的效果，在你的图像处理程序中可以完成这个操作。

凹凸贴图的值因该在50%灰度图上波动(RGB=127,127,127), 亮的值代表凸起，暗的值代表凹陷。

凹凸贴图可以为纹理图大小的1/4，而不会影响外观效果。

现在你应该对这篇文章中内容的大概有了一个基本的认识，希望你读的愉快。

如果你有任何纹理，请联系我或访问我的网站<http://www.glhint.de>

我必须感谢以下的人：

Michael I. Gold , 它写出了凹凸贴图的原理

Diego Tártara , 它写出了示例代码

NVidia 公司 , 他在Internet发布了大量的源码

最后感谢Nehe , 它对我的OpenGL学习起了很大的帮助



## 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

## 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

## 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

## 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

## 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。



## 第23课



球面映射:

这一个将教会你如何把环境纹理包裹在你的3D模型上，让它看起来象反射了周围的场景一样。

球体环境映射是一个创建快速金属反射效果的方法，但它并不像真实世界里那么精确！我们从18课的代码开始来创建这个教程，教你如何创建这种效果。

在我们开始之前，看一下红宝书中的介绍。它定义球体环境映射为一幅位于无限远的图像，把它映射到球面上。

在Photoshop中创建一幅球体环境映射图。

首先，你需要一幅球体环境映射图，用来把它映射到球体上。在Photoshop中打开一幅图并选择所有的像素，创建它的一个复制。

接着，我们把图像变为2的幂次方大小，一般为128x128或256x256。

最后使用扭曲(distort)滤镜，并应用球体效果。然后把它保存为\*.bmp文件。

我们并没有添加任何全局变量，只是把纹理组的大小变为6，以保存6幅纹理。

// 保存6幅纹理

下面我们要做的就是载入这些纹理

```
int LoadGLTextures()
{
    int Status=FALSE;

    AUX_RGBImageRec *TextureImage[2]; // 创建纹理的保存空间

    memset(TextureImage,0,sizeof(void *)*2); // 清空为0

    // 载入*.bmp图像
    if ((TextureImage[0]=LoadBMP("Data/BG.bmp")) && // 背景图
        (TextureImage[1]=LoadBMP("Data/Reflect.bmp"))) // 反射图(球形纹理图)
    {
        Status=TRUE;

        glGenTextures(6, &texture[0]); // 创建6个纹理

        for (int loop=0; loop<=1; loop++)
        {
            // 创建临近点过滤纹理图
            glBindTexture(GL_TEXTURE_2D, texture[loop]); // 创建纹理0和1
            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, TextureImage[loop]->sizeX, TextureImage[loop]->sizeY,
0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);

            // 创建线形过滤纹理图
            glBindTexture(GL_TEXTURE_2D, texture[loop+2]); // 创建纹理2,3
            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, TextureImage[loop]->sizeX, TextureImage[loop]->sizeY,
0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);

            // 创建线形Mipmap纹理图
        }
    }
}
```

```

glBindTexture(GL_TEXTURE_2D, texture[loop+4]);      // 创建纹理4 , 5
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, 3, TextureImage[loop]->sizeX, TextureImage
[loop]->sizeY,
                      GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
}
for (loop=0; loop<=1; loop++)
{
if (TextureImage[loop])                                // 如果图像存在则清除
{
    if (TextureImage[loop]->data)
    {
        free(TextureImage[loop]->data);
    }
    free(TextureImage[loop]);
}
}
}

return Status;
}

```

我们对立方体的绘制代码做了一些小的改动，把法线的范围从[-1,1]缩放到[-0.5,0.5]。如果法向量太大的话，会产生一些块状效果，影响视觉效果。

```

GLvoid glDrawCube()
{
    glBegin(GL_QUADS);
    // 前面
    glNormal3f( 0.0f, 0.0f, 0.5f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
    // 背面
    glNormal3f( 0.0f, 0.0f,-0.5f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
    glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);

```

```

glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// 上面
glNormal3f( 0.0f, 0.5f, 0.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
// 下面
glNormal3f( 0.0f,-0.5f, 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);
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
glNormal3f( 0.5f, 0.0f, 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);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
// 左面
glNormal3f(-0.5f, 0.0f, 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);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();
}

```

在初始化OpenGL中，我们添加一些新的函数来使用球体纹理映射。  
下面的代码让OpenGL自动为我们计算使用球体映射时，顶点的纹理坐标。

```

glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);           // 设置s方向的纹理
自动生成
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);           // 设置t方向的纹
理自动生成

```

我们几乎完成了所有的工作！接下来要做的就是就是绘制渲染，我删除了一些二次几何体，因为它们的视觉效果并不好。当然我们需要OpenGL为这些几何体自动生成坐标，接着选择球体映射纹理并绘制几何体。最后把OpenGL状态设置正常模式。

```
int DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity(); // 重置视口

    glTranslatef(0.0f,0.0f,z);

    glEnable(GL_TEXTURE_GEN_S); // 自动生成s方向纹理坐标
    glEnable(GL_TEXTURE_GEN_T); // 自动生成t方向纹理坐标

    glBindTexture(GL_TEXTURE_2D, texture[filter+(filter+1)]); // 绑定纹理
    glPushMatrix();
    glRotatef(xrot,1.0f,0.0f,0.0f);
    glRotatef(yrot,0.0f,1.0f,0.0f);
    switch(object)
    {
        case 0:
            glDrawCube();
            break;
        case 1:
            glTranslatef(0.0f,0.0f,-1.5f); // 创建圆柱
            gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32);
            break;
        case 2:
            gluSphere(quadratic,1.3f,32,32); // 创建球
            break;
        case 3:
            glTranslatef(0.0f,0.0f,-1.5f); // 创建圆锥
            gluCylinder(quadratic,1.0f,0.0f,3.0f,32,32);
            break;
    };
    glPopMatrix();
    glDisable(GL_TEXTURE_GEN_S); // 禁止自动生成纹理坐标
    glDisable(GL_TEXTURE_GEN_T);

    xrot+=xspeed;
```

```
yrot+=yspeed;  
return TRUE; // 成功返回  
}
```

最后我们使用空格来切换各个不同的几何体

```
if (keys[' '] && !sp)  
{  
    sp=TRUE;  
    object++;  
    if(object>3)  
        object=0;  
}
```

我们成功了！现在你可以使用环境映射纹理做一些非常棒的特效了。我想做一个立方体环境映射的例子，但我现在的显卡不支持这种特效，所以只有等到以后了。

谢谢，并祝你好运！

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

#### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容,大多都来自共享的



资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第22课

第24课 &gt;



## 第24课

```

Renderer: GeForce Ti 4600/AGP/8MB
Vendor: NVIDIA Corporation
Version: 1.4.0

1 GL_ARB_depth_texture
2 GL_ARB_imaging
3 GL_ARB_multisample
4 GL_ARB_multitexture
5 GL_ARB_point_parameters
6 GL_ARB_shadow
7 GL_ARB_texture_border_clamp
8 GL_ARB_texture_compression
9 GL_ARB_texture_cube_map

NeHe Productions
  
```

扩展，剪裁和TGA图像文件的加载：

在这一课里，你将学会如何读取你显卡支持的OpenGL的扩展，并在你指定的剪裁区域把它显示出来。

这个教程有一些难度，但它会让你学到很多东西。我听到很多朋友问我扩展方面的内容和怎样找到它们。这个教程将交给你这一切。

我将教会你怎样滚动屏幕的一部分和怎样绘制直线，最重要的是从这一课起，我们将不使用AUX库，以及\*.bmp文件。我将告诉你如何使用Targa(TGA)图像文件。因为它简单并且支持alpha通道，它可以使你更容易的创建酷的效果。

接下来我们要做的第一件事就是不包含glaux.h头文件和glaux.lib库。另外，在使用glaux库时，经常会发生一些可疑的警告，现在我们可以测定告别它了。

```

#include <stdarg.h>           // 处理可变参数的函数的头文件
#include <string.h>            // 处理字符串的头文件
  
```

接下来我们添加一些变量，第一个为滚动参数。第二给变量记录扩展的个数，swidth和sheight记录剪切矩形的大小。base为字体显示列表的开始值。

```
int      scroll;           // 用来滚动屏幕
int      maxtokens;        // 保存扩展的个数
int      swidth;           // 剪裁宽度
int      sheight;          // 剪裁高度
GLuint   base;             // 字符显示列表的开始值
```

现在我们创建一个数据结构用来保存TGA文件，接着我们使用这个结构来加载纹理。

```
typedef struct           // 创建加载TGA图像文件结构
{
    GLubyte *imageData;  // 图像数据指针
    GLuint bpp;          // 每个数据所占的位数（必须为24或32）
    GLuint width;         // 图像宽度
    GLuint height;        // 图像高度
    GLuint texID;         // 纹理的ID值
} TextureImage;           // 结构名称

TextureImage textures[1]; // 保存一个纹理
```

这个部分的代码将要加载一个TGA文件并把它转换为纹理。必须注意的是这部分代码只能加载24/32位的不压缩的TGA文件。

这个函数包含两个参数，一个保存载入的图像，一个为将载入的文件名。

TGA文件包含一个12个字节的文件头，载入图像后，我们用type来设置图像中像素格式在OpenGL中的对应。如果是24位的图像我们使用GL\_RGB，如果是32位的图像我们使用GL\_RGBA。

```
bool LoadTGA(TextureImage *texture, char *filename)           // 把TGA文件加载入内存
{
    GLubyte  TGAheader[12]={0,0,2,0,0,0,0,0,0,0,0,0};           // 无压缩的TGA文件头
```

```

GLubyte    TGAcompare[12];           // 保存读入的文件头信息
GLubyte    header[6];               // 保存最有用的图像信息，宽，高，位深
GLuint     bytesPerPixel;          // 记录每个颜色所占用的字节数
GLuint     imageSize;              // 记录文件大小
GLuint     temp;                  // 临时变量
GLuint     type=GL_RGBA;           // 设置默认的格式为GL_RGBA，即32位图像

```

下面这个函数读取TGA文件，并记录文件信息。TGA文件格式如下所示：

Tga图像格式  
无颜色表 rgb 图像

偏移	长度	描述	32位常用图像文件各个字节的值
0	1	指出图像信息字段的长度，其取值范围是0到255，当它为0时表示没有图像的信息字段。	0
1	1	是否使用颜色表，0表示没有颜色表，1表示颜色表存在	0
2	1	该字段总为2。图像类型码，tga一共有6种格式，2表示无颜色表rgb图像	2
3			0
4			0
5	5	颜色表规格，总为0。	0
6			0
7			0

### 8 10 图像规格说明开始

8	2	图像x坐标起始位置，一般为0	0
9			
10	2	图像y坐标起始位置，一般为0	0
11			
12	2	图像宽度，以像素为单位	256
13			
14	2	图像高度，以像素为单位	256
15			

16	1	图像每像素存储占用位 (bit) 数	32
17	1	图像描述符字节 bits 3-0 - 每像素对应的属性位的位数，对于 TGA 24， 该值为 0 bit 4 - 保留，必须为 0 bit 5 - 屏幕起始位置标志，0 = 原点在左下角，1 = 原 点在左上角 一般这个字节设为0x00即可	00100000 <sub>(2)</sub>
18	可变	图像数据域 这里存储了 (宽度) x (高度) 个像素，每个像素中 的 rgb 色值该色值包含整数个字节	...

如果一切顺利，读取文件后关闭文件。

```

FILE *file = fopen(filename, "rb"); // 打开一个TGA文件

if( file==NULL || // 文件存在么?
    fread(TGAc,1,sizeof(TGAc),file)!=sizeof(TGAc) || // 是否包含12个字节
    memcmp(header,TGAc,sizeof(TGAc))!=0 || // 是否是我们需要的格
    // 式?
    fread(header,1,sizeof(header),file)!=sizeof(header)) // 如果是读取下面六个图像信息
{
    if (file == NULL) // 文件不存在返回错误
        return false;
    else
    {
        fclose(file); // 关闭文件返回错误
        return false;
    }
}

```

下面的代码记录文件的宽度和高度，并判断文件是否为24位/32位TGA文件。

```

texture->width = header[1] * 256 + header[0]; // 记录文件高度
texture->height = header[3] * 256 + header[2]; // 记录文件宽度

```

```

if( texture->width <=0 || // 宽度是否小于0
    texture->height <=0 || // 高度是否小于0
    (header[4]!=24 && header[4]!=32)) // TGA文件是24/32位 ?
{
    fclose(file); // 如果失败关闭文件，返回错误
    return false;
}

```

下面的代码记录文件的位深和加载它需要的内存大小

```

texture->bpp = header[4]; // 记录文件的位深
bytesPerPixel = texture->bpp/8; // 记录每个象素所占的字节数
imageSize = texture->width*texture->height*bytesPerPixel; // 计算TGA文件加载所需要的内存大小

```

下面的代码为图像数据分配内存并载入它

```

texture->imageData=(GLubyte *)malloc(imageSize); // 分配内存去保存TGA数据

if( texture->imageData==NULL || // 系统是否分配了足够的内存 ?
    fread(texture->imageData, 1, imageSize, file)!=imageSize) // 是否成功读入内存?
{
    if(texture->imageData!=NULL) // 是否有数据被加载
        free(texture->imageData); // 如果是，则释放载入的数据

    fclose(file); // 关闭文件
    return false; // 返回错误
}

```

TGA文件中，颜色的存储顺序为BGR，而OpenGL中颜色的顺序为RGB，所以我们需要交换每个象素中R和B的值。如果一切顺利，TGA文件中的图像数据将按照OpenGL的要求存储在内存中了。

```

for(GLuint i=0; i<int(imageSize); i+=bytesPerPixel)           // 循环所有的像素
{
    temp=texture->imageData[i];
    texture->imageData[i] = texture->imageData[i + 2];
    texture->imageData[i + 2] = temp;
}

fclose (file);                                              // 关闭文件

```

下面的代码创建一个纹理，并设置过滤方式为线性

```

// 创建纹理
glGenTextures(1, &texture[0].texID);                      // 创建纹理，并记录纹理ID

glBindTexture(GL_TEXTURE_2D, texture[0].texID);            // 绑定纹理
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); // 设置过
滤器为线性过滤
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);

```

判断图像的位数是否为24，如果是则设置类型为GL\_RGB

```

if (texture[0].bpp==24)                                     // 是否为24位图像？
{
    type=GL_RGB;                                           // 如果是设置类型为GL_RGB
}

```

下面的代码在OpenGL中创建一个纹理

```

glTexImage2D(GL_TEXTURE_2D, 0, type, texture[0].width, texture[0].height, 0, type,
GL_UNSIGNED_BYTE, texture[0].imageData);

return true;                                // 纹理绑定完成，成功返回
}

```

下面的代码是从图像创建字体的典型的方法，这些代码将包含在后面的课程中，以显示文字。

只有一个不同的地方，纹理0用来保存字符图像。

```

GLvoid BuildFont(GLvoid)                      // 创建字体显示列表
{
    base=glGenLists(256);                      // 创建256个显示列表
    glBindTexture(GL_TEXTURE_2D, textures[0].texID); // 绑定纹理
    for (int loop1=0; loop1<256; loop1++)        // 循环创建256个显示列表
    {
        float cx=float(loop1%16)/16.0f;          // 当前字符的X位置
        float cy=float(loop1/16)/16.0f;           // 当前字符的Y位置

        glNewList(base+loop1,GL_COMPILE);          // 开始创建显示列表
        glBegin(GL_QUADS);                      // 创建一个四边形用来包含字符图像
        glTexCoord2f(cx,1.0f-cy-0.0625f);        // 左下方纹理坐标
        glVertex2d(0,16);                        // 左下方坐标
        glTexCoord2f(cx+0.0625f,1.0f-cy-0.0625f); // 右下方纹理坐标
        glVertex2i(16,16);                       // 右下方坐标
        glTexCoord2f(cx+0.0625f,1.0f-cy-0.001f); // 右上方纹理坐标
        glVertex2i(16,0);                        // 右上方坐标
        glTexCoord2f(cx,1.0f-cy-0.001f);         // 左上方纹理坐标
        glVertex2i(0,0);                         // 左上方坐标
        glEnd();                                // 四边形创建完毕
        glTranslated(14,0,0);                   // 向右移动14个单位
    glEndList();                             // 结束创建显示列表
}

```

下面的函数用来删除显示字符的显示列表

```
GLvoid KillFont(GLvoid)
{
    glDeleteLists(base,256); // 从内存中删除256个显示列表
}
```

glPrint函数只有一点变化，我们在Y轴方向把字符拉长一倍

```
GLvoid glPrint(GLint x, GLint y, int set, const char *fmt, ...)
{
    char text[1024]; // 保存我们的字符
    va_list ap; // 指向第一个参数

    if (fmt == NULL) // 如果要显示的字符为空则返回
        return;

    va_start(ap, fmt); // 开始分析参数，并把结果写入到text中
    vsprintf(text, fmt, ap);
    va_end(ap);

    if (set>1) // 如果字符集大于1则使用第二个字符集
    {
        set=1;
    }

    glEnable(GL_TEXTURE_2D); // 使用纹理映射
    glLoadIdentity(); // 重置视口矩阵
    glTranslated(x,y,0); // 平移到(x,y,0)处
    glListBase(base-32+(128*set)); // 选择字符集

    glScalef(1.0f,2.0f,1.0f); // 沿Y轴放大一倍

    glCallLists(strlen(text),GL_UNSIGNED_BYTE, text); // 把字符写入到屏幕
    glDisable(GL_TEXTURE_2D); // 禁止纹理映射
}
```

窗口改变大小的函数使用正投影，把视口范围设置为(0,0)-(640,480)

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
{
    swidth=width;                                // 设置剪切矩形为窗口大小
    sheight=height;
    if (height==0)                               // 防止高度为0时，被0除
    {
        height=1;
    }
    glViewport(0,0,width,height);                // 设置窗口可见区
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0f,640,480,0.0f,-1.0f,1.0f);      // 设置视口大小为640x480
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
```

初始化操作非常简单，我们载入字体纹理，并创建字符显示列表，如果顺利，则成功返回。

```
int InitGL(GLvoid)
{
    if (!LoadTGA(&textures[0],"Data/Font.TGA"))          // 载入字体纹理
    {
        return false;                                     // 载入失败则返回
    }

    BuildFont();                                       // 创建字体

    glShadeModel(GL_SMOOTH);                          // 使用平滑着色
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);              // 设置黑色背景
    glClearDepth(1.0f);                               // 设置深度缓存中的值为1
    glBindTexture(GL_TEXTURE_2D, textures[0].texID);   // 绑定字体纹理
```

```

    return TRUE;                                // 成功返回
}

```

绘制代码几乎是全新的:) , token为一个指向字符串的指针 , 它将保存OpenGL扩展的全部字符串 , cnt纪录扩展的个数。

接下来清楚背景 , 并显示OpenGL的销售商 , 实现它的公司和当前的版本。

```

int DrawGLScene(GLvoid)
{
    char *token;                                // 保存扩展字符串
    int cnt=0;                                   // 纪录扩展字符串的个数

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);           // 清楚背景和
深度缓存

    glColor3f(1.0f,0.5f,0.5f);                   // 设置为红色
    glPrint(50,16,1,"Renderer");
    glPrint(80,48,1,"Vendor");
    glPrint(66,80,1,"Version");

```

下面的代码显示OpenGL实现方面的相关信息 , 完成之后我们用蓝色在屏幕的下方写上 “ NeHe Productions ” , 当然你可以使用任何你想使用的字符 , 比如"DancingWind Translate"。

```

    glColor3f(1.0f,0.7f,0.4f);                  // 设置为橘黄色
    glPrint(200,16,1,(char *)glGetString(GL_RENDERER));      // 显示OpenGL的实现组织
    glPrint(200,48,1,(char *)glGetString(GL_VENDOR));        // 显示销售商
    glPrint(200,80,1,(char *)glGetString(GL_VERSION));       // 显示当前版本

    glColor3f(0.5f,0.5f,1.0f);                   // 设置为蓝色
    glPrint(192,432,1,"NeHe Productions");          // 在屏幕的底端写上NeHe Productions字
串

```

现在我们绘制显示扩展名的白色线框方块，并用一个更大的白色线框方块把所有的内容包围起来。

```
glLoadIdentity(); // 重置模型变换矩阵
glColor3f(1.0f,1.0f,1.0f); // 设置为白色
glBegin(GL_LINE_STRIP);
    glVertex2d(639,417);
    glVertex2d( 0,417);
    glVertex2d( 0,480);
    glVertex2d(639,480);
    glVertex2d(639,128);
glEnd();
glBegin(GL_LINE_STRIP);
    glVertex2d( 0,128);
    glVertex2d(639,128);
    glVertex2d(639, 1);
    glVertex2d( 0, 1);
    glVertex2d( 0,417);
glEnd();
```

glScissor函数用来设置剪裁区域，如果启用了GL\_SCISSOR\_TEST,绘制的内容只能在剪裁区域中显示。

下面的代码设置窗口的中部为剪裁区域，并获得扩展名字符串。

```
glScissor(1 ,int(0.135416f*sheight),swidth-2,int(0.597916f*sheight)); // 定义剪裁区域
glEnable(GL_SCISSOR_TEST); // 使用剪裁测试

char* text=(char*)malloc(strlen((char *)glGetString(GL_EXTENSIONS))+1); // 为保存OpenGL
扩展的字符串分配内存空间
strcpy (text,(char *)glGetString(GL_EXTENSIONS)); // 返回OpenGL扩展字符串
```

下面我们创建一个循环，循环显示每个扩展名，并纪录扩展名的个数

// 按空格分割text字符串，并把分割后的字符

```

token=strtok(text," ");
while(token!=NULL)
{
    cnt++;
    if (cnt>maxtokens)
    {
        maxtokens=cnt;
    }
}

```

// 增加计数器

// 纪录最大的扩展名数量

现我们已经获得第一个扩展名，下一步我们把它显示在屏幕上。

我们已经显示了三行文本，它们在Y轴上占用了 $3 \times 32 = 96$ 个像素的宽度，所以我们显示的第一个行文本的位置是(0,96)，一次类推第*i*行文本的位置是 $(0,96 + (cnt * 32))$ ，但我们需要考虑当前滚动过的位置，默认为向上滚动，所以我们得到显示第*i*行文本的位置为 $(0,96 + (cnt * 32) - scroll)$ 。

当然它们不会都显示出来，记得我们使用了剪裁，只显示 $(0,96) - (0,96 + 32 * 9)$ 之间的文本，其它的都被剪裁了。

更具我们上面的讲解，显示的第一个行如下：

1 GL\_ARB\_multitexture

```

glColor3f(0.5f,1.0f,0.5f);           // 设置颜色为绿色
glPrint(0.96+(cnt*32)-scroll,0,"%i",cnt); // 绘制第几个扩展名

glColor3f(1.0f,1.0f,0.5f);           // 设置颜色为黄色
glPrint(50,96+(cnt*32)-scroll,0,token); // 输出第i个扩展名

```

当我们显示完所有的扩展名,我们需要检查一下是否已经分析完了所有的字符串。我们使用strtok(NULL," ")函数代替strtok(text," ")函数，把第一个参数设置为NULL会检查当前指针位置到字符串末尾是否包含" "字符，如果包含返回其位置，否则返回NULL。

我们举例说明上面的过程，例如字符串"GL\_ARB\_multitexture GL\_EXT\_abgr GL\_EXT\_bgra",它是以空格分割字符串的，第一次调用strtok("text"," ")返回text的首位置，并在空格" "的位置加入一个NULL。以后每次调用，删除NULL，返回空格位置的下一个位置，接着搜索下一个空格的位置，并在空格的位置加入一个NULL。直道返回NULL。

返回NULL时循环停止，表示已经显示完所有的扩展名。

```
token=strtok(NULL," "); // 查找下一个扩展名  
}
```

下面的代码让OpenGL返回到默认的渲染状态，并释放分配的内存资源

```
glDisable(GL_SCISSOR_TEST); // 禁用剪裁测试  
free (text); // 释放分配的内存
```

下面的代码让OpenGL完成所有的任务，并返回TRUE

```
glFlush(); // 执行所有的渲染命令  
return TRUE; // 成功返回  
}
```

KillGLWindow函数基本没有变化，唯一改变的是需要删除我们创建的字体

```
KillFont();
```

```
// 删除字体
```

CreateGLWindow(), 和 WndProc() 函数保持不变

在WinMain()函数中我们需要加入新的按键控制

下面的代码检查向上的箭头是否被按下，如果scroll大于0，我们把它减少2

```
if (keys[VK_UP] && (scroll>0))          // 向上的箭头是否被按下?  
{  
    scroll-=2;                          // 如果是，减少scroll的值  
}
```

如果向下的箭头被按住，并且scroll小于32\*(maxtoken-9),则增加scroll的值，32是每一个字符的高度，9是可以显示的行数。

```
if (keys[VK_DOWN] && (scroll<32*(maxtokens-9))) // 向下的箭头是否被按住  
{  
    scroll+=2;                          // 如果是，增加scroll的值  
}
```

我希望你觉得这个教程有趣，学完了这个教程你应该知道如何获得你的显卡的发售商的名称，实现OpenGL的组织和你的显卡所使用的OpenGL的版本。进一步，你应该知道你的显卡支持的扩展的名称，并熟练的使用剪切矩形和加载TGA图像。

如果你发现任何问题，请让我知道。我想做最好的教程，你的反馈对我很重要。



### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来

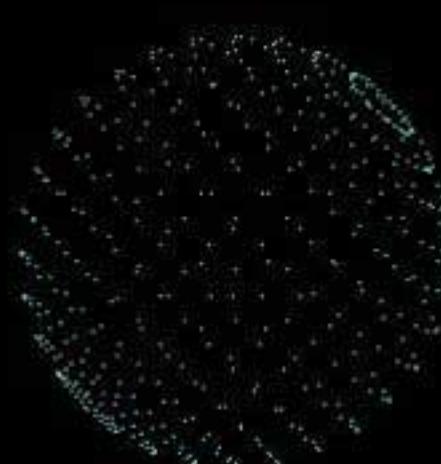
给我买书了，她真的是我见过的最好的女孩，希望  
我能带给她幸福。

< 第23课

第25课 >



## 第25课



变形和从文件中加载3D物体:

在这一课中，你将学会如何从文件加载3D模型，并且平滑的从一个模型变换为另一个模型。

欢迎来到这激动人心的一课，在这一课里，我们将介绍模型的变形。需要注意的是各个模型必须要有相同的顶点，才能一一对应，并应用变形。  
在这一课里，我们同样要教会你如何从一个文件中读取模型数据。  
文件开始的部分和前面一样，没有任何变化。

我们接下来添加几个旋转变量，用来记录旋转的信息。并使用cx,cy,cz设置物体在屏幕上  
的位置。  
变量key用来记录当前的模型，step用来设置相邻变形之间的中间步骤。如step为200，则  
需要200次，才能把一个物体变为另一个物体。  
最后我们用一个变量来设置是否使用变形。

```
GLfloat xrot,yrot,zrot,  
xspeed,yspeed,zspeed,  
cx,cy,cz=-15;  
// X, Y & Z 轴的旋转角度  
// X, Y & Z 轴的旋转速度  
// 物体的位置
```

```

int      key=1;           // 物体的标识符
int      step=0,steps=200; // 变换的步数
bool    morph=FALSE;     // 是否使用变形

```

下面的结构定义一个三维顶点

```

typedef struct
{
    float x, y, z;
} VERTEX;

```

下面的结构使用顶点来描述一个三维物体

```

typedef struct          // 物体结构
{
    int      verts;      // 物体中顶点的个数
    VERTEX   *points;    // 包含顶点数据的指针
} OBJECT;

```

maxver用来记录各个物体中最大的顶点数，如一个物体使用5个顶点，另一个物体使用20个顶点，那么物体的顶点个数为20。

接下来定义了四个我们使用的模型物体，并把相邻模型变形的中间状态保存在helper中，sour保存原模型物体，dest保存将要变形的模型物体。

```

int      maxver;          // 最大的顶点数
OBJECT   morph1,morph2,morph3,morph4, // 我们的四个物体
        helper,*sour,*dest; // 帮助物体,原物体 , 目标物体

```

## WndProc()函数没有变化

下面的函数用来为模型分配保存顶点数据的内存空间

```
void objallocate(OBJECT *k,int n)
{
    k->points=(VERTEX*)malloc(sizeof(VERTEX)*n);           // 分配n个顶点的内存空间
}
```

下面的函数用来释放为模型分配的内存空间

```
void objfree(OBJECT *k)
{
    free(k->points);
}
```

下面的代码用来读取文件中的一行。

我们用一个循环来读取字符，最多读取255个字符，当遇到'\n'回车时，停止读取并立即返回。

```
void readstr(FILE *f,char *string)                      // 读取一行字符
{
    do
    {
        fgets(string, 255, f);                          // 最多读取255个字符
    } while ((string[0] == '/') || (string[0] == '\n')); // 遇到回车则停止读取
    return;                                              // 返回
}
```

下面的代码用来加载一个模型文件，并为模型分配内存，把数据存储进去。

```
void objload(char *name,OBJECT *k) // 从文件加载一个模型
{
    int ver; // 保存顶点个数
    float rx,ry,rz; // 保存模型位置
    FILE *filein; // 打开的文件句柄
    char oneline[255]; // 保存255个字符

    filein = fopen(name, "rt"); // 打开文本文件，供读取

    readstr(filein, oneline); // 读入一行文本
    sscanf(oneline, "Vertices: %d\n", &ver); // 搜索字符串"Vertices: "，并把其后的
    // 顶点数保存在ver变量中
    k->verts=ver; // 设置模型的顶点个数
    objallocate(k,ver); // 为模型数据分配内存
}
```

下面的循环，读取每一行（即每个顶点）的数据，并把它保存到内存中？

```
for (int i=0;i<ver;i++)
{
    readstr(filein, oneline); // 循环所有的顶点
    sscanf(oneline, "%f %f %f", &rx, &ry, &rz); // 读取一行数据
    // 把顶点数据保存在rx,ry,rz中

    k->points[i].x = rx; // 保存当前顶点的x坐标
    k->points[i].y = ry; // 保存当前顶点的y坐标
    k->points[i].z = rz; // 保存当前顶点的z坐标
}
fclose(filein); // 关闭文件

if(ver>maxver) maxver=ver; // 记录最大的顶点数
}
```

下面的函数根据设定的间隔，计算第i个顶点每次变换的位移

```
VERTEX calculate(int i) // 计算第i个顶点每次变换的位移
{
    VERTEX a;
    a.x=(sour->points[i].x-dest->points[i].x)/steps;
    a.y=(sour->points[i].y-dest->points[i].y)/steps;
    a.z=(sour->points[i].z-dest->points[i].z)/steps;
    return a;
}
```

ReSizeGLScene()函数没有变化

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
```

下面的函数完成初始化功能，它设置混合模式为半透明

```
int InitGL(GLvoid)
{
    glBlendFunc(GL_SRC_ALPHA,GL_ONE); // 设置半透明混合模式
    glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // 设置清除色为黑色
    glClearDepth(1.0); // 设置深度缓存中值为1
    glDepthFunc(GL_LESS); // 设置深度测试函数
    glEnable(GL_DEPTH_TEST); // 启用深度测试
    glShadeModel(GL_SMOOTH); // 设置着色模式为光滑着色
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
```

下面的代码用来加载我们的模型物体

```
maxver=0;                                // 初始化最大顶点数为0
objload("data/sphere.txt",&morph1);        // 加载球模型
objload("data/torus.txt",&morph2);          // 加载圆环模型
objload("data/tube.txt",&morph3);           // 加载立方体模型
```

第四个模型不从文件读取，我们在 (-7, -7, -7) - (7, 7, 7) 之间随机生成模型点,它和我们载入的模型都一样具有486个顶点。

```
objallocate(&morph4,486);                  // 为第四个模型分配内存资源
for(int i=0;i<486;i++)                     // 随机设置486个顶点
{
    morph4.points[i].x=((float)(rand()%14000)/1000)-7;
    morph4.points[i].y=((float)(rand()%14000)/1000)-7;
    morph4.points[i].z=((float)(rand()%14000)/1000)-7;
}
```

初始化中间模型为球体，并把原和目标模型都设置为球

```
objload("data/sphere.txt",&helper);
sour=dest=&morph1;

return TRUE;                                // 初始化完成，成功返回
}
```

下面是具体的绘制代码，向往常一样我们先设置模型变化，以便我们更好的观察。

```
void DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);      // 清空缓存
    glLoadIdentity();                                         // 重置模型变换矩阵
```

```

glTranslatef(cx,cy,cz);           // 平移和旋转
glRotatef(xrot,1,0,0);
glRotatef(yrot,0,1,0);
glRotatef(zrot,0,0,1);

xrot+=xspeed; yrot+=yspeed; zrot+=zspeed;      // 根据旋转速度 , 增加旋转角度

GLfloat tx,ty,tz;                // 顶点临时变量
VERTEX q;                        // 保存中间计算的临时顶点

```

接下来我们来绘制模型中的点，如果启用了变形，则计算变形的中间过程点。

```

glBegin(GL_POINTS);             // 点绘制开始
for(int i=0;i<morph1.verts;i++) // 循环绘制模型1中的每一个顶点
{
    if(morph) q=calculate(i); else q.x=q.y=q.z=0;      // 如果启用变形 , 则计算中间模
型
    helper.points[i].x-=q.x;
    helper.points[i].y-=q.y;
    helper.points[i].z-=q.z;
    tx=helper.points[i].x;                            // 保存计算结果到x,y,z变量中
    ty=helper.points[i].y;
    tz=helper.points[i].z;

```

为了让动画看起来流畅，我们一共绘制了三个中间状态的点。让变形过程从蓝绿色向蓝色下一个状态变化。

```

glColor3f(0,1,1);               // 设置颜色
glVertex3f(tx,ty,tz);          // 绘制顶点
glColor3f(0,0.5f,1);           // 把颜色变蓝一些
tx-=2*q.x; ty-=2*q.y; ty-=2*q.y; // 如果启用变形 , 则绘制2步后的顶点
glVertex3f(tx,ty,tz);
glColor3f(0,0,1);               // 把颜色变蓝一些
tx-=2*q.x; ty-=2*q.y; ty-=2*q.y; // 如果启用变形 , 则绘制2步后的顶点
glVertex3f(tx,ty,tz);

```

```
    }  
    glEnd();  
    // 绘制结束
```

最后如果启用了变形，则增加递增的步骤参数，然后绘制下一个点。

```
// 如果启用变形则把变形步数增加  
if(morph && step<=steps)step++; else { morph=FALSE; sour=dest; step=0;}  
  
return TRUE; // 一切OK  
}
```

KillGLWindow() 函数基本没有变化，只是添加释放5个模型内存的代码

```
objfree(&morph1);  
        // 释放模型1内存  
objfree(&morph2);  
        // 释放模型2内存  
objfree(&morph3);  
        // 释放模型3内存  
objfree(&morph4);  
        // 释放模型4内存  
objfree(&helper);  
        // 释放模型5内存
```

CreateGLWindow() 函数没有变化

```
BOOL CreateGLWindow()
```

```
LRESULT CALLBACK WndProc()
```

在WinMain()函数中，我们添加了一些键盘控制的函数

```

if(keys[VK_PRIOR])
    zspeed+=0.01f;                                // PageUp键是否被按下
                                                // 按下增加绕z轴旋转的速度

if(keys[VK_NEXT])
    zspeed-=0.01f;                                // PageDown键是否被按下
                                                // 按下减少绕z轴旋转的速度

if(keys[VK_DOWN])
    xspeed+=0.01f;                                // 下方向键是否被按下
                                                // 按下增加绕x轴旋转的速度

if(keys[VK_UP])
    xspeed-=0.01f;                                // 上方向键是否被按下
                                                // 按下减少绕x轴旋转的速度

if(keys[VK_RIGHT])
    yspeed+=0.01f;                                // 右方向键是否被按下
                                                // 按下增加沿y轴旋转的速度

if(keys[VK_LEFT])
    yspeed-=0.01f;                                // 左方向键是否被按下
                                                // 按下减少沿y轴旋转的速度

if (keys['Q'])
    cz-=0.01f;                                   // Q键是否被按下
                                                // 是则向屏幕里移动

if (keys['Z'])
    cz+=0.01f;                                   // Z键是否被按下
                                                // 是则向屏幕外移动

if (keys['W'])
    cy+=0.01f;                                   // W键是否被按下
                                                // 是则向上移动

if (keys['S'])
    cy-=0.01f;                                   // S键是否被按下
                                                // 是则向下移动

if (keys['D'])
    cx+=0.01f;                                   // D键是否被按下
                                                // 是则向右移动

if (keys['A'])
    cx-=0.01f;                                   // A键是否被按下
                                                // 是则向左移动

```

1 , 2 , 3 , 4键用来设置变形的目标模型

```
if (keys['1'] && (key!=1) && !morph)           // 如果1被按下，则变形到模型1
{
    key=1;
    morph=TRUE;
    dest=&morph1;
}
if (keys['2'] && (key!=2) && !morph)           // 如果2被按下，则变形到模型1
{
    key=2;
    morph=TRUE;
    dest=&morph2;
}
if (keys['3'] && (key!=3) && !morph)           // 如果3被按下，则变形到模型1
{
    key=3;
    morph=TRUE;
    dest=&morph3;
}
if (keys['4'] && (key!=4) && !morph)           // 如果4被按下，则变形到模型1
{
    key=4;
    morph=TRUE;
    dest=&morph4;
}
```

我希望你能喜欢这个教程，相信你已经学会了变形动画。

Piotr Cieslak 的代码非常的新颖，希望通过这个教程你能知道如何从文件中加载三维模型。

这份教程化了我三天的时间，如果有任何错误请告诉我。



## 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

## 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

## 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

## 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

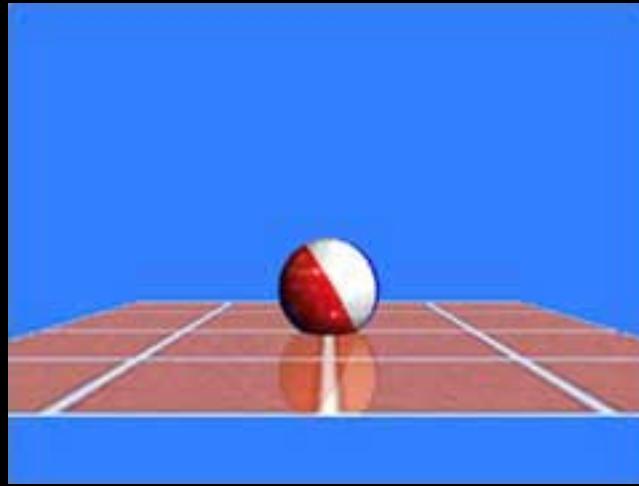
## 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。



## 第26课



剪裁平面，蒙板缓存和反射:

在这一课中你将学会如何创建镜面显示效果，它使用剪裁平面，蒙板缓存等OpenGL中一些高级的技巧。

欢迎来到另一个激动人心的课程，这课的代码是Banu Cosmin所写，当然教程还是我自己写的。在这课里，我将教你创建真正的反射，基于物理的。由于它将用到蒙板缓存，所以需要耗费一些资源。当然随着显卡和CPU的发展，这些都不是问题了，好了让我们开始吧！

下面我们设置光源的参数

```
static GLfloat LightAmb[] = {0.7f, 0.7f, 0.7f, 1.0f};           // 环境光  
static GLfloat LightDif[] = {1.0f, 1.0f, 1.0f, 1.0f};           // 漫射光  
static GLfloat LightPos[] = {4.0f, 4.0f, 6.0f, 1.0f};           // 灯光的位置
```

下面用二次几何体创建一个球，并设置纹理

```
GLUquadricObj *q; // 使用二次几何体创建球

GLfloat xrot = 0.0f; // X方向的旋转角度
GLfloat yrot = 0.0f; // Y方向的旋转角的
GLfloat xrotspeed = 0.0f; // X方向的旋转速度
GLfloat yrotspeed = 0.0f; // Y方向的旋转速度
GLfloat zoom = -7.0f; // 移入屏幕7个单位
GLfloat height = 2.0f; // 球离开地板的高度

GLuint texture[3]; // 使用三个纹理
```

ReSizeGLScene() 和LoadBMP() 没有变化

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)
```

```
AUX_RGBImageRec *LoadBMP(char *Filename)
```

下面的代码载入纹理

```
int LoadGLTextures() // 载入*.bmp文件，并转化为纹理
{
    int Status=FALSE;
    AUX_RGBImageRec *TextureImage[3]; // 创建三个图象
    memset(TextureImage,0,sizeof(void *)*3);
    if ((TextureImage[0]=LoadBMP("Data/EnvWall.bmp")) && // 载入地板图像
        (TextureImage[1]=LoadBMP("Data/Ball.bmp")) && // 载入球图像
        (TextureImage[2]=LoadBMP("Data/EnvRoll.bmp"))) // 载入强的图像
    {
        Status=TRUE;
        glGenTextures(3, &texture[0]); // 创建纹理
    }
}
```

```

for (int loop=0; loop<3; loop++)           // 循环设置三个纹理参数
{
    glBindTexture(GL_TEXTURE_2D, texture[loop]);
    glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage[loop]->sizeX, TextureImage[loop]->sizeY, 0, GL_RGB, GL_UNSIGNED_BYTE, TextureImage[loop]->data);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
}
for (loop=0; loop<3; loop++)
{
    if (TextureImage[loop])
    {
        if (TextureImage[loop]->data)
        {
            free(TextureImage[loop]->data);
        }
        free(TextureImage[loop]);
    }
}
return Status;
}

```

一个新的函数glClearStencil被加入到初始化代码中，它用来设置清空操作后蒙板缓存中的值。其他的操作保持不变。

```

int InitGL(GLvoid)           // 初始化OpenGL
{
    if (!LoadGLTextures())      // 载入纹理
    {
        return FALSE;
    }
    glShadeModel(GL_SMOOTH);
    glClearColor(0.2f, 0.5f, 1.0f, 1.0f);
    glClearDepth(1.0f);
    glClearStencil(0);          // 设置蒙板值
    glEnable(GL_DEPTH_TEST);
    glDepthFunc(GL_LEQUAL);
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
    glEnable(GL_TEXTURE_2D);     // 使用2D纹理
}

```

下面的代码用来启用光照

```
glLightfv(GL_LIGHT0, GL_AMBIENT, LightAmb);
glLightfv(GL_LIGHT0, GL_DIFFUSE, LightDif);
glLightfv(GL_LIGHT0, GL_POSITION, LightPos);
glEnable(GL_LIGHT0);
glEnable(GL_LIGHTING);
```

下面的代码使用二次几何体创建一个球体，在前面的教程中都已经详纤，这里不再重  
复。

```
q = gluNewQuadric();                                // 创建一个二次几何体
gluQuadricNormals(q, GL_SMOOTH);                  // 使用平滑法线
gluQuadricTexture(q, GL_TRUE);                     // 使用纹理
glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP); // 设置球纹理
映射
glTexGeni(GL_T, GL_TEXTURE_GEN_MODE, GL_SPHERE_MAP);

return TRUE;                                         // 初始化完成，成功返回
}
```

下面的代码绘制我们的球

```
void DrawObject()                                     // 绘制我们的球
{
    glColor3f(1.0f, 1.0f, 1.0f);                   // 设置为白色
    glBindTexture(GL_TEXTURE_2D, texture[1]);        // 设置为球的纹理
    gluSphere(q, 0.35f, 32, 16);                   // 绘制球
```

绘制完一个白色的球后，我们使用环境贴图来绘制另一个球，把这两个球按alpha混合起来。

```

glBindTexture(GL_TEXTURE_2D, texture[2]);           // 设置为环境纹理
glColor4f(1.0f, 1.0f, 1.0f, 0.4f);                 // 使用alpha为40%的白色
glEnable(GL_BLEND);                                // 启用混合
glBlendFunc(GL_SRC_ALPHA, GL_ONE);                  // 把原颜色的40%与目标颜色混合
glEnable(GL_TEXTURE_GEN_S);                         // 使用球映射
glEnable(GL_TEXTURE_GEN_T);

gluSphere(q, 0.35f, 32, 16);                      // 绘制球体，并混合

glDisable(GL_TEXTURE_GEN_S);                        // 让OpenGL回到默认的属性
glDisable(GL_TEXTURE_GEN_T);
glDisable(GL_BLEND);
}

```

## 绘制地板

```

void DrawFloor()
{
    glBindTexture(GL_TEXTURE_2D, texture[0]);           // 选择地板纹理，地板由一个长方
形组成
    glBegin(GL_QUADS);
        glNormal3f(0.0, 1.0, 0.0);
        glTexCoord2f(0.0f, 1.0f);                      // 左下
        glVertex3f(-2.0, 0.0, 2.0);

        glTexCoord2f(0.0f, 0.0f);                      // 左上
        glVertex3f(-2.0, 0.0, -2.0);

        glTexCoord2f(1.0f, 0.0f);                      // 右上
        glVertex3f( 2.0, 0.0, -2.0);

        glTexCoord2f(1.0f, 1.0f);                      // 右下
        glVertex3f( 2.0, 0.0, 2.0);
    glEnd();
}

```

```
    glVertex3f( 2.0, 0.0, 2.0);
    glEnd();
}
```

现在到了我们绘制函数的地方，我们将把所有的模型结合起来创建一个反射的场景。向往常一样先把各个缓存清空，接着定义我们的剪切平面，它用来剪切我们的图像。这个平面的方程为`equ[] = {0,-1,0,0}`,向你所看到的它的法线是指向-y轴的，这告诉我们你只能看到y轴坐标小于0的像素，如果你启用剪切功能的话。

关于剪切平面，我们在后面会做更多的讨论。继续吧：）

```
int DrawGLScene(GLvoid)
{
    // 清除缓存
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

    // 设置剪切平面
    double eqr[] = {0.0f, -1.0f, 0.0f, 0.0f};
```

下面我们把地面向下平移0.6个单位，因为我们的眼睛在 $y=0$ 的平面，如果不平移的话，那么看上去平面就会变为一条线，为了看起来更真实，我们平移了它。

```
glLoadIdentity();
glTranslatef(0.0f, -0.6f, zoom); // 平移和缩放地面
```

下面我们设置了颜色掩码，在默认情况下所有的颜色都可以写入，即在函数`glColorMask`中，所有的参数都被设为`GL_TRUE`，如果设为零表示这部分颜色不可写入。现在我们不希望在屏幕上绘制任何东西，所以把参数设为0。

```
glColorMask(0,0,0,0);
```

下面来设置蒙板缓存和蒙板测试。

首先我们启用蒙板测试，这样就可以修改蒙板缓存中的值。

下面我们来解释蒙板测试函数的含义：

当你使用glEnable(GL\_STENCIL\_TEST)启用蒙板测试之后，蒙板函数用于确定一个颜色片段是应该丢弃还是保留（被绘制）。蒙板缓存区中的值与参考值ref进行比较，比较标准是func所指定的比较函数。参考值和蒙板缓存区的值都可以与掩码进行AND操作。蒙板测试的结果还导致蒙板缓存区根据glStencilOp函数所指定的行为进行修改。

func的参数值如下：

常量	含义
GL_NEVER	从不通过蒙板测试
GL_ALWAYS	总是通过蒙板测试
GL_LESS	只有参考值<(蒙板缓存区的值&mask)时才通过
GL_EQUAL	只有参考值<=(蒙板缓存区的值&mask)时才通过
GL_GREATER	只有参考值=(蒙板缓存区的值&mask)时才通过
GL_NOTEQUAL	只有参考值>=(蒙板缓存区的值&mask)时才通过
GL_NOTEQUAL	只有参考值!=(蒙板缓存区的值&mask)时才通过

接下来我们解释glStencilOp函数，它用来根据比较结果修改蒙板缓存区中的值，它的函数原形为：

void glStencilOp(GLenum sfail, GLenum zfail, GLenum zpass)，各个参数的含义如下：

sfail

当蒙板测试失败时所执行的操作

zfail

当蒙板测试通过，深度测试失败时所执行的操作

zpass

当蒙板测试通过，深度测试通过时所执行的操作

具体的操作包括以下几种

常量	描述
GL_KEEP	保持当前的蒙板缓存区值
GL_ZERO	把当前的蒙板缓存区值设为0
GL_REPLACE	用glStencilFunc函数所指定的参考值替换蒙板参数值

GL_INCR	增加当前的蒙板缓存区值，但限制在允许的范围内
GL_DECR	减少当前的蒙板缓存区值，但限制在允许的范围内
GL_INVERT	将当前的蒙板缓存区值进行逐位的翻转

当完成了以上操作后我们绘制一个地面，当然现在你什么也看不到，它只是把覆盖地面的蒙板缓存区中的相应位置设为1。

```
glEnable(GL_STENCIL_TEST);           // 启用蒙板缓存
glStencilFunc(GL_ALWAYS, 1, 1);      // 设置蒙板测试总是通过，参考值设为1，掩码值也设
为1
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE); // 设置当深度测试不通过时，保留蒙板
中的值不变。如果通过则使用参考值替换蒙板值
glDisable(GL_DEPTH_TEST);           // 禁用深度测试
DrawFloor();                         // 绘制地面
```

我们现在已经在蒙板缓存区中建立了地面的蒙板了，这是绘制影子的关键，如果想知道为什么，接着向后看吧:)

下面我们启用深度测试和绘制颜色，并相应设置蒙板测试和函数的值，这种设置可以使我们在屏幕上绘制而不改变蒙板缓存区的值。

```
glEnable(GL_DEPTH_TEST);           //启用深度测试
glColorMask(1,1,1,1);             //可以绘制颜色
glStencilFunc(GL_EQUAL, 1, 1);     //下面的设置指定当我们绘制时，不改变蒙板
缓存区的值
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);
```

下面的代码设置并启用剪切平面，使得只能在地面的下方绘制

```
glEnable(GL_CLIP_PLANE0);          // 使用剪切平面
glClipPlane(GL_CLIP_PLANE0, eqr);  // 设置剪切平面为地面，并设置它的法
```

线为向下

```
glPushMatrix();           // 保存当前的矩阵
glScalef(1.0f, -1.0f, 1.0f); // 沿Y轴反转
```

由于上面已经启用了蒙板缓存，则你只能在蒙板中值为1的地方绘制，反射的实质就是在反射屏幕的对应位置在绘制一个物体，并把它放置在反射平面中。下面的代码完成这个功能

```
glLightfv(GL_LIGHT0, GL_POSITION, LightPos);          // 设置灯光0
glTranslatef(0.0f, height, 0.0f);
glRotatef(xrot, 1.0f, 0.0f, 0.0f);
glRotatef(yrot, 0.0f, 1.0f, 0.0f);
DrawObject();                                         // 绘制反射的球
glPopMatrix();                                       // 弹出保存的矩阵
glDisable(GL_CLIP_PLANE0);                         // 禁用剪切平面
glDisable(GL_STENCIL_TEST);                        // 关闭蒙板
```

下面的代码绘制地面，并把地面颜色和反射的球颜色混合，使其看起来像反射的效果。

```
glLightfv(GL_LIGHT0, GL_POSITION, LightPos);
glEnable(GL_BLEND);                                // 启用混合
glDisable(GL_LIGHTING);                           // 关闭光照
	glColor4f(1.0f, 1.0f, 1.0f, 0.8f);             // 设置颜色为白色
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // 设置混合系数
DrawFloor();                                         // 绘制地面
```

下面的代码在距地面高为height的地方绘制一个真正的球

```
glEnable(GL_LIGHTING);                            // 使用光照
glDisable(GL_BLEND);                           // 禁用混合
glTranslatef(0.0f, height, 0.0f);               // 移动高位height的位置
```

```
glRotatef(xrot, 1.0f, 0.0f, 0.0f); // 设置球旋转的角度  
glRotatef(yrot, 0.0f, 1.0f, 0.0f);  
DrawObject(); // 绘制球
```

下面的代码用来处理键盘控制等常规操作

```
xrot += xrotspeed; // 更新X轴旋转速度  
yrot += yrotspeed; // 更新Y轴旋转速度  
glFlush(); // 强制OpenGL执行所有命令  
return TRUE; // 成功返回  
}
```

下面的代码处理键盘控制，上下左右控制球的旋转。PageUp/PageDown控制球的上下。  
A , Z控制球离你的远近。

```
void ProcessKeyboard()  
{  
    if (keys[VK_RIGHT])    yrotspeed += 0.08f;  
    if (keys[VK_LEFT])     yrotspeed -= 0.08f;  
    if (keys[VK_DOWN])     xrotspeed += 0.08f;  
    if (keys[VK_UP])       xrotspeed -= 0.08f;  
  
    if (keys['A'])         zoom += 0.05f;  
    if (keys['Z'])         zoom -= 0.05f;  
  
    if (keys[VK_PRIOR])   height += 0.03f;  
    if (keys[VK_NEXT])    height -= 0.03f;  
}
```

KillGLWindow() 函数没有任何改变

## GLvoid KillGLWindow(GLvoid)

CreateGLWindow()函数基本没有改变，只是添加了以行启用蒙板缓存

```
static PIXELFORMATDESCRIPTOR pfd=
{
    sizeof(PIXELFORMATDESCRIPTOR),
    1,
    PFD_DRAW_TO_WINDOW |
    PFD_SUPPORT_OPENGL |
    PFD_DOUBLEBUFFER,
    PFD_TYPE_RGBA,
    bits,
    0, 0, 0, 0, 0,
    0,
    0,
    0,
    0, 0, 0, 0,
    16,
```

下面就是在这个函数中唯一改变的地方，记得把0变为1，它启用蒙板缓存。

```
    1, // 使用蒙板缓存
    0,
    PFD_MAIN_PLANE,
    0,
    0, 0, 0
};
```

WinMain()函数基本没有变化，只是加上以行键盘控制的处理函数

```
ProcessKeyboard(); // 处理按键相应
```

我真的希望你能喜欢这个教程，我清楚地知道我想做的每一件事，以及如何一步一步实现我心中想创建的效果。但把它表达出来又是另一回事，当你坐下来并实际的去向那些从来没听到过蒙板缓存的人解释这一切时，你就会清楚了。好了，如果你有什么不清楚的，或者有更好的建议，请让我知道，我想些最好的教程，你的反馈很重要！

### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照

顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第25课

第27课 >



## 第27课



影子:

这是一个高级的主题，请确信你已经熟练的掌握了基本的OpenGL，并熟悉蒙板缓存。当然它会给你留下深刻的印象的。

欢迎来到另一个有些复杂的课程，阴影。这一课的效果好的有些让人不可思议，阴影可以变形，混合在其他的物体上。

这一课要求你必须对OpenGL比较了解，它假设你知道许多OpenGL的知识，你必须知道蒙板缓存，基本的OpenGL步骤。如果你对这些不太熟悉，我建议你可以看看前面的教程。当然，在这一课里，我们用到了很多数学知识，请准备好一本数学手册在你的身边。

首先我们定义阴影体可以延伸的距离。

```
// 定义阴影体可以延伸的距离  
#define INFINITY    100
```

下面定义一个3D顶点结构

```
// 3D顶点结构
struct sPoint
{
    GLfloat x, y, z;
};
```

定义一个平面结构

```
// 平面方程为: ax + by + cz + d = 0
struct sPlaneEq
{
    GLfloat a, b, c, d;
};
```

下面定义一个用来投影的三角形的结构

3个整形索引指定了模型中三角形的三个顶点  
 第二个变量指定了三角形面的法线  
 平面方程描述了三角所在的平面  
 临近的3个顶点索引，指定了与这个三角形相邻的三个顶点  
 最后一个变量指定这个三角形是否投出阴影

// 描述一个模型表面的结构

```
struct sPlane
{
    unsigned int p[3];           // 3个整形索引指定了模型中三角形的三个顶点
    sPoint normals[3];          // 第二个变量指定了三角形面的法线
    unsigned int neigh[3];       // 与本三角形三个边相邻的面的索引
    sPlaneEq PlaneEq;          // 平面方程描述了三角所在的平面
    bool visible;               // 最后一个变量指定这个三角形是否投出阴影?
};
```

最后我们用下面的结构描述一个产生阴影的物体。

```
struct glObject{  
    GLuint nPlanes, nPoints;  
    sPoint points[100];  
    sPlane planes[200];  
};
```

下面的代码用来读取模型，它的代码本身就解释了它的功能。它从文件中读取数据，并把顶点和索引存储在上面定义的结构中，并把所有的临近顶点初始化为-1，它代表这没有任何顶点与它相邻，我们将在以后计算它。

```
bool readObject( const char *filename, glObject*o)  
{  
    FILE *file;  
    unsigned int i;  
  
    file = fopen(st, "r");  
    if (!file) return FALSE;  
    //读取顶点  
    fscanf(file, "%d", &(o->nPoints));  
    for (i=1;i<=o->nPoints;i++){  
        fscanf(file, "%f", &(o->points[i].x));  
        fscanf(file, "%f", &(o->points[i].y));  
        fscanf(file, "%f", &(o->points[i].z));  
    }  
    //读取三角形面  
    fscanf(file, "%d", &(o->nPlanes));  
    for (i=0;i<nPlanes;i++){  
        fscanf(file, "%d", &(o->planes[i].p[0]));  
        fscanf(file, "%d", &(o->planes[i].p[1]));  
        fscanf(file, "%d", &(o->planes[i].p[2]));  
        //读取每个顶点的法线  
        fscanf(file, "%f", &(o->planes[i].normals[0].x));  
        fscanf(file, "%f", &(o->planes[i].normals[0].y));  
    }  
}
```

```

fscanf(file, "%f", &(o->planes[i].normals[0].z));
fscanf(file, "%f", &(o->planes[i].normals[1].x));
fscanf(file, "%f", &(o->planes[i].normals[1].y));
fscanf(file, "%f", &(o->planes[i].normals[1].z));
fscanf(file, "%f", &(o->planes[i].normals[2].x));
fscanf(file, "%f", &(o->planes[i].normals[2].y));
fscanf(file, "%f", &(o->planes[i].normals[2].z));
}
return true;
}

```

现在从setConnectivity函数开始,事情变得越来越复杂了,这个函数用来查找每个面的相邻的顶点,下面是它的伪代码:

对于模型中的每一个面A

    对于面A中的每一条边

        如果我们不只到这条边相邻的顶点

            那么对于模型中除了面A外的每一个面B

                对于面B中的每一条边

                    如果面A的边和面B的边是同一条边,那么这两个面相邻

                        设置面A和面B的相邻属性

下面的代码完成上面伪代码中最后两行的内容,你先获得每个面中边的两个顶点,然后检测他们是否相邻,如果是则设置各自的相邻顶点信息

```

int vertA1 = pFaceA->vertexIndices[edgeA];
int vertA2 = pFaceA->vertexIndices[(edgeA+1)%3];

int vertB1 = pFaceB->vertexIndices[edgeB];
int vertB2 = pFaceB->vertexIndices[(edgeB+1)%3];

// 测试他们是否为同一边,如果是则设置相应的相邻顶点信息
if ((vertA1 == vertB1 && vertA2 == vertB2) || (vertA1 == vertB2 && vertA2 == vertB1))
{
    pFaceA->neighbourIndices[edgeA] = faceB;
}

```

```
pFaceB->neighbourIndices[edgeB] = faceA;  
edgeFound = true;  
break;  
}  
}
```

完整的SetConnectivity函数的代码如下

```
// 设置相邻顶点信息  
inline void SetConnectivity(glObject *o){  
    unsigned int p1i, p2i, p1j, p2j;  
    unsigned int P1i, P2i, P1j, P2j;  
    unsigned int i,j,ki,kj;  
  
    //对于模型中的每一个面A  
    for(i=0;inPlanes-1;i++)  
    {  
        //对于除了此面的其它的面B  
        for(j=i+1;jnPlanes;j++)  
        {  
            //对于面A中的每一个相邻的顶点  
            for(ki=0;ki<3;ki++)  
            {  
                //如果这个相邻的顶点没有被设置  
                if(!o->planes[i].neigh[ki])  
                {  
                    for(kj=0;kj<3;kj++)  
                    {  
                        p1i=ki;  
                        p1j=kj;  
                        p2i=(ki+1)%3;  
                        p2j=(kj+1)%3;  
  
                        p1i=o->planes[i].p[p1i];  
                        p2i=o->planes[i].p[p2i];  
                        p1j=o->planes[j].p[p1j];  
                        p2j=o->planes[j].p[p2j];  
  
                        //如果面A的边P1i->P1j和面B的边P2i->P2j为同一条边，则又下面的公式的  
                }  
            }  
        }  
    }  
}
```

下面的函数用来绘制模型

```
// 绘制模型，像以前一样它绘制组成模型的三角形
void drawObject( const ShadowedObject& object )
{
    glBegin( GL_TRIANGLES );
    for ( int i = 0; i < object.nFaces; i++ )
    {
        const Face& face = object.pFaces[i];
        for ( int j = 0; j < 3; j++ )
        {
            const Point3f& vertex = object.pVertices[face.vertexIndices[j]];
            glNormal3f( face.normals[j].x, face.normals[j].y, face.normals[j].z );
            glVertex3f( vertex.x, vertex.y, vertex.z );
        }
    }
    glEnd();
}
```

下面的函数用来计算平面的方程参数

```
void calculatePlane( const ShadowedObject& object, Face& face )
{
    // 获得平面的三个顶点
    const Point3f& v1 = object.pVertices[face.vertexIndices[0]];
    const Point3f& v2 = object.pVertices[face.vertexIndices[1]];
    const Point3f& v3 = object.pVertices[face.vertexIndices[2]];

    face.planeEquation.a = v1.y*(v2.z-v3.z) + v2.y*(v3.z-v1.z) + v3.y*(v1.z-v2.z);
    face.planeEquation.b = v1.z*(v2.x-v3.x) + v2.z*(v3.x-v1.x) + v3.z*(v1.x-v2.x);
    face.planeEquation.c = v1.x*(v2.y-v3.y) + v2.x*(v3.y-v1.y) + v3.x*(v1.y-v2.y);
    face.planeEquation.d = -( v1.x*( v2.y*v3.z - v3.y*v2.z ) +
        v2.x*(v3.y*v1.z - v1.y*v3.z) +
        v3.x*(v1.y*v2.z - v2.y*v1.z) );
}
```

你还可以呼吸么?好的,我们继续:) 接下来你将学习如何去投影,castShadow函数几乎用到了所有OpenGL的功能,完成这个函数后,把它传递到doShadowPass函数来通过两个渲染通道绘制出阴影.

首先,我们看看哪些面面对着灯光,我们可以通过灯光位置和平面方程计算出.如果灯光到平面的位置大于0,则位于灯光的上方,否则位于灯光的下方(如果有什么问题,翻一下你高中的解析几何).

```
void castShadow( ShadowedObject& object, GLfloat *lightPosition )
{
    // 设置哪些面在灯光的前面
    for ( int i = 0; i < object.nFaces; i++ )
    {
        const Plane& plane = object.pFaces[i].planeEquation;

        GLfloat side = plane.a*lightPosition[0] +
            plane.b*lightPosition[1] +
            plane.c*lightPosition[2] +

```

```

plane.d;

if ( side > 0 )
    object.pFaces[i].visible = true;
else
    object.pFaces[i].visible = false;
}

```

下面设置必要的状态来渲染阴影.

首先,禁用灯光和绘制颜色,因为我们不计算光照,这样可以节约计算量.

接着,设置深度缓存,深度测试还是需要的,但我们不希望我们的阴影体向实体一样具有深度,所以关闭深度缓存.

最后我们启用蒙板缓存,让阴影体的位置在蒙板中被设置为1.

```

glDisable( GL_LIGHTING );           // 关闭灯光
glColorMask( GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE ); // 关闭颜色缓存的写入
glDepthFunc( GL_LEQUAL );          // 设置深度比较函数
glDepthMask( GL_FALSE );           // 关闭深度缓存的写入
glEnable( GL_STENCIL_TEST );        // 使用蒙板缓存
glStencilFunc( GL_ALWAYS, 1, 0xFFFFFFFF ); // 设置蒙板函数

```

现在到了阴影被实际渲染得地方了,我们使用了下面提到的doShadowPass函数,它用来绘制阴影体的边界面.我们通过两个步骤来绘制阴影体,首先使用前向面增加阴影体在蒙板缓存中的值,接着使用后向面减少阴影体在蒙板缓存中的值.

```

// 如果是逆时针 (即面向视点) 的多边形, 通过了蒙板和深度测试, 则把蒙板的值增加1
glFrontFace( GL_CCW );
glStencilOp( GL_KEEP, GL_KEEP, GL_INCR );
doShadowPass( object, lightPosition );
// 如果是顺时针 (即背向视点) 的多边形, 通过了蒙板和深度测试, 则把蒙板的值减少1
glFrontFace( GL_CW );
glStencilOp( GL_KEEP, GL_KEEP, GL_DECR );
doShadowPass( object, lightPosition );

```

为了更好的理解这两个步骤,我建议你把第二步注释掉看看效果,如下所示:



图 1: 步骤1



图 2: 步骤2

最后一步就是把阴影体所在的位置绘制上阴影的颜色

```
glFrontFace( GL_CCW );
glColorMask( GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE );

// 把阴影绘制上颜色
glColor4f( 0.0f, 0.0f, 0.0f, 0.4f );
glEnable( GL_BLEND );
glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA );
glStencilFunc( GL_NOTEQUAL, 0, 0xFFFFFFFF );
glStencilOp( GL_KEEP, GL_KEEP, GL_KEEP );
glPushMatrix();
glLoadIdentity();
glBegin( GL_TRIANGLE_STRIP );
    glVertex3f(-0.1f, 0.1f,-0.10f);
    glVertex3f(-0.1f,-0.1f,-0.10f);
    glVertex3f( 0.1f, 0.1f,-0.10f);
    glVertex3f( 0.1f,-0.1f,-0.10f);
glEnd();
glPopMatrix();
}
```

下面的部分我们绘制构成阴影体边界的四边形,当我们循环所有的三角形面的时候,我们检测它是否是边界边,如果是我们绘制从灯光到这个边界边的射线,并衍生它用来构成四边形.

这里要用一个蛮力,我们检测物体模型中每一个三角形面,找出其边界并连接灯光到边界的直线,把直线延长出一定的距离,构成阴影体.

下面的代码完成这些功能,它看起来并没有想象的复杂.

```
void doShadowPass(glObject *o, float *lp)
{
    unsigned int i, j, k, jj;
    unsigned int p1, p2;
    sPoint      v1, v2;

    //对模型中的每一个面
    for (i=0; inPlanes;i++)
    {
        //如果面在灯光的前面
        if (o->planes[i].visible)
        {
            //对于被灯光照射的面的每一个相邻的面
            for (j=0;j<3;j++)
            {
                k = o->planes[i].neigh[j];
                //如果面不存在 , 或不被灯光照射 , 那么这个边是边界
                if ((!k) || (!o->planes[k-1].visible))
                {
                    // 获得面的两个顶点
                    p1 = o->planes[i].p[j];
                    jj = (j+1)%3;
                    p2 = o->planes[i].p[jj];

                    //计算边的顶点到灯光的方向 , 并放大100倍
                    v1.x = (o->points[p1].x - lp[0])*100;
                    v1.y = (o->points[p1].y - lp[1])*100;
                    v1.z = (o->points[p1].z - lp[2])*100;

                    v2.x = (o->points[p2].x - lp[0])*100;
                    v2.y = (o->points[p2].y - lp[1])*100;
                    v2.z = (o->points[p2].z - lp[2])*100;
```

```

//绘制构成阴影体边界的面
glBegin(GL_TRIANGLE_STRIP);
    glVertex3f(o->points[p1].x,
               o->points[p1].y,
               o->points[p1].z);
    glVertex3f(o->points[p1].x + v1.x,
               o->points[p1].y + v1.y,
               o->points[p1].z + v1.z);

    glVertex3f(o->points[p2].x,
               o->points[p2].y,
               o->points[p2].z);
    glVertex3f(o->points[p2].x + v2.x,
               o->points[p2].y + v2.y,
               o->points[p2].z + v2.z);
    glEnd();
}
}
}

}

```

既然我们已经能绘制阴影了,那么我们开始绘制我们的场景吧

```

bool drawGLScene()
{
    GLmatrix16f Minv;
    GLvector4f wlp, lp;

    // 清空缓存
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

    glLoadIdentity();                                // 设置灯光，并绘制球
    glTranslatef(0.0f, 0.0f, -20.0f);
    glLightfv(GL_LIGHT1, GL_POSITION, LightPos);
    glTranslatef(SpherePos[0], SpherePos[1], SpherePos[2]);

```

```
gluSphere(q, 1.5f, 32, 16);
```

下面我们计算灯光在物体坐标系中的位置

```
glLoadIdentity();
glRotatef(-yrot, 0.0f, 1.0f, 0.0f);
glRotatef(-xrot, 1.0f, 0.0f, 0.0f);
glTranslatef(-ObjPos[0], -ObjPos[1], -ObjPos[2]);
glGetFloatv(GL_MODELVIEW_MATRIX, Minv);           // 计算从世界坐标系变化到物体坐
标系中的坐标
lp[0] = LightPos[0];                                // 保存灯光的位置
lp[1] = LightPos[1];
lp[2] = LightPos[2];
lp[3] = LightPos[3];
VMatMult(Minv, lp);                                // 计算最后灯光的位置
```

下面绘制房间，物体和它的阴影

```
glLoadIdentity();
glTranslatef(0.0f, 0.0f, -20.0f);
DrawGLRoom();                                     // 绘制房间
glTranslatef(ObjPos[0], ObjPos[1], ObjPos[2]);
glRotatef(xrot, 1.0f, 0.0f, 0.0f);
glRotatef(yrot, 0.0f, 1.0f, 0.0f);
DrawGLObject(obj);                               // 绘制物体
CastShadow(&obj, lp);                           // 绘制物体的阴影
```

下面的代码绘制一个黄色的球代表了灯光的位置

```
	glColor4f(0.7f, 0.4f, 0.0f, 1.0f);
```

```
glDisable(GL_LIGHTING);
glDepthMask(GL_FALSE);
glTranslatef(lp[0], lp[1], lp[2]);
gluSphere(q, 0.2f, 16, 8);
glEnable(GL_LIGHTING);
glDepthMask(GL_TRUE);
```

## 最后设置物体的控制

```
xrot += xspeed; // 增加X轴选择速度
yrot += yspeed; // 增加Y轴选择速度

glFlush(); // 强制OpenGL完成所有的命令
return TRUE; // 成功返回
}
```

## 绘制房间墙面

```
void DrawGLRoom() // 绘制房间(盒装)
{
    glBegin(GL_QUADS); // 绘制四边形
    // 地面
    glNormal3f(0.0f, 1.0f, 0.0f); // 法线向上
    glVertex3f(-10.0f, -10.0f, -20.0f);
    glVertex3f(-10.0f, -10.0f, 20.0f);
    glVertex3f( 10.0f, -10.0f, 20.0f);
    glVertex3f( 10.0f, -10.0f, -20.0f);
    // 天花板
    glNormal3f(0.0f, -1.0f, 0.0f); // 法线向下
    glVertex3f(-10.0f, 10.0f, 20.0f);
    glVertex3f(-10.0f, 10.0f, -20.0f);
    glVertex3f( 10.0f, 10.0f, -20.0f);
    glVertex3f( 10.0f, 10.0f, 20.0f);
    // 前面
    glNormal3f(0.0f, 0.0f, 1.0f); // 法线向后
    glVertex3f(-10.0f, -10.0f, -20.0f);
    glVertex3f( 10.0f, -10.0f, -20.0f);
    glVertex3f( 10.0f, 10.0f, -20.0f);
    glVertex3f(-10.0f, 10.0f, -20.0f);
}
```

```

glVertex3f(-10.0f, 10.0f,-20.0f);
glVertex3f(-10.0f,-10.0f,-20.0f);
glVertex3f( 10.0f,-10.0f,-20.0f);
glVertex3f( 10.0f, 10.0f,-20.0f);
// 后面
glNormal3f(0.0f, 0.0f,-1.0f);           // 法线向前
glVertex3f( 10.0f, 10.0f, 20.0f);
glVertex3f( 10.0f,-10.0f, 20.0f);
glVertex3f(-10.0f,-10.0f, 20.0f);
glVertex3f(-10.0f, 10.0f, 20.0f);
// 左面
glNormal3f(1.0f, 0.0f, 0.0f);           // 法线向右
glVertex3f(-10.0f, 10.0f, 20.0f);
glVertex3f(-10.0f,-10.0f, 20.0f);
glVertex3f(-10.0f,-10.0f,-20.0f);
glVertex3f(-10.0f, 10.0f,-20.0f);
// 右面
glNormal3f(-1.0f, 0.0f, 0.0f);          // 法线向左
glVertex3f( 10.0f, 10.0f,-20.0f);
glVertex3f( 10.0f,-10.0f,-20.0f);
glVertex3f( 10.0f,-10.0f, 20.0f);
glVertex3f( 10.0f, 10.0f, 20.0f);
glEnd();                                // 结束绘制
}

```

下面的函数完成矩阵M与向量V的乘法 $M=M*V$

```

void VMatMult(GLmatrix16f M, GLvector4f v)
{
    GLfloat res[4];                      // 保存中间计算结果
    res[0]=M[ 0]*v[0]+M[ 4]*v[1]+M[ 8]*v[2]+M[12]*v[3];
    res[1]=M[ 1]*v[0]+M[ 5]*v[1]+M[ 9]*v[2]+M[13]*v[3];
    res[2]=M[ 2]*v[0]+M[ 6]*v[1]+M[10]*v[2]+M[14]*v[3];
    res[3]=M[ 3]*v[0]+M[ 7]*v[1]+M[11]*v[2]+M[15]*v[3];
    v[0]=res[0];                         // 把结果保存在V中
    v[1]=res[1];
    v[2]=res[2];
    v[3]=res[3];
}

```

}

下面的函数用来初始化模型对象

```
int InitGLObjets() // 初始化模型对象
{
    if (!ReadObject("Data/Object2.txt", &obj)) // 读取模型数据
    {
        return FALSE; // 返回失败
    }

    SetConnectivity(&obj); // 设置相邻顶点的信息

    for ( int i=0;i < obj.nPlanes;i++)
        CalcPlane(obj, &obj.planes[i]); // 计算每个面的平面参数

    return TRUE; //成功返回
}
```

其他的函数我们不做过多解释了,这会分散你的注意力,好好享受阴影带给你的快感吧.

下面还有一些说明:

球体不会产生阴影,因为我们没有设置其投影.

如果你发现程序很慢,买块好的显卡吧.

最后我希望你喜欢它,如果有什么好的建议,请告诉我.



## 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

## 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

## 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

## 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

## 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。



## 第28课



贝塞尔曲面:

这是一课关于数学运算的，没有别的内容了。来，有信心就看看它吧。

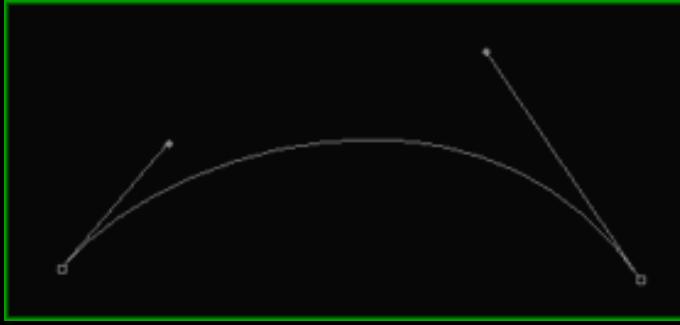
### 贝塞尔曲面

作者: David Nikdel ([ogapo@ithink.net](mailto:ogapo@ithink.net))

这篇教程旨在介绍贝塞尔曲面，希望有比我更懂艺术的人能用她作出一些很COOL的东东并且展示给大家。教程不能用做一个完整的贝塞尔曲面库，而是一个展示概念的程序让你熟悉曲面怎样实现的。而且这不是一篇正规的文章，为了方便理解，我也许在有些地方术语不当；我希望大家能适应这个。最后，对那些已经熟悉贝塞尔曲面想看我写的如何的，真是丢脸；- ) 但你要是找到任何纰漏让我或者NeHe知道，毕竟人无完人嘛？还有，所有代码没有象我一般写程序那样做优化，这是故意的。我想每个人都能明白写的是什么。好，我想介绍到此为止，继续看下文！

数学：：恶魔之音：：（警告：内容有点长~）

好，如果想理解贝塞尔曲面没有对其数学基本的认识是很难的，如果你不愿意读这一部分或者你已经知道了关于她的数学知识你可以跳过。首先我会描述贝塞尔曲线再介绍生成贝塞尔曲面。奇怪的是，如果你用过一个图形程序，你就已经熟悉了贝塞尔曲线，也许你接触的是另外的名称。它们是画曲线的最基本的方法，而且通常被表示成一系列点，其中有两个点与两端点表示左右两端的切线。下图展示了一个例子。



这是最基础的贝塞尔曲线（长点的由很多点在一起（多到你都没发现））。这个曲线由4个点定义，有2个端点和2个中间控制点。对计算机而言这些点都是一样的，但是特意的我们通常把前后两对点分别连接，因为他们的连线与短点相切。曲线是一个参数化曲线，画的时候从曲线上平均找几点连接。这样你可以控制曲线曲面的精度（和计算量）。最通常的方法是远距离少细分近距离多细分，对视点，看上去总是很完好的曲面而对速度的影响总是最小。

贝塞尔曲面基于一个基本方程，其他复杂的都是基于此。方程为：

$$t + (1 - t) = 1$$

看起来很简单不是？的确是的，这是最基本的贝塞尔曲线，一个一维的曲线。你也许从术语中猜到，贝塞尔曲线是多项式形式的。从线性代数知，一个一维的多项式是一条直线，没多大意思。好，因为基本方程对所有t都成立，我们可以平方，立方两边，怎么都行，等式都是成立的，对吧？好，我们试试立方。

$$(t + (1-t))^3 = 1^3$$

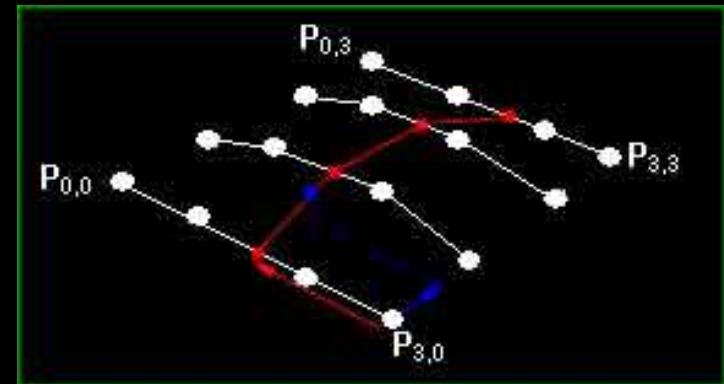
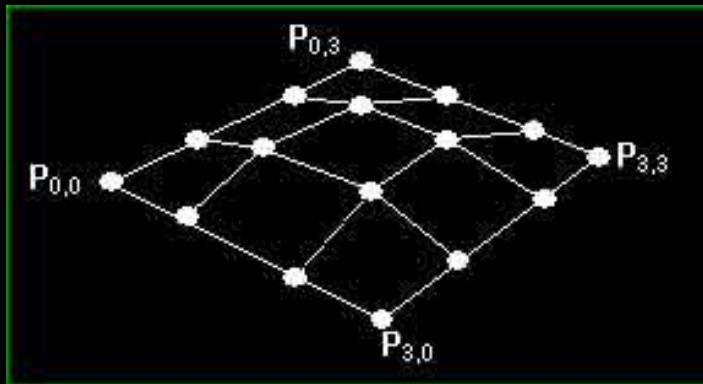
$$t^3 + 3*t^2*(1-t) + 3*t*(1-t)^2 + (1-t)^3 = 1$$

这是我们最常用的计算贝塞尔曲面的方程，a)她是最低维的不需要在一个平面内的多项式（有4个控制点），而且b)两边的切线互相没有联系（对于2维的只有3个控制点）。那么你看到了贝塞尔曲线了吗？呵呵，我们都没有，因为我还要加一个东西。

好，因为方程左边等于1，可以肯定如果你把所有项加起来还是等于1。这是否意味着在计算曲线上一点时可以以此决定该用每个控制点的多少呢？（答案是肯定的）你对了！当我们要计算曲线上一点的值我们只需要用控制点（表示为向量）乘以每部分再加起来。基本上我们要用 $0 \leq t \leq 1$ ，但不是必要的。不明白了把？这里有函数：

$$P1*t^3 + P2*3*t^2*(1-t) + P3*3*t*(1-t)^2 + P4*(1-t)^3 = P_{new}$$

因为多项式是连续的，有一个很好的办法在4个点间插值。曲线仅经过 $P_1, P_4$ ，分别当 $t=0, 1$ 。好，一切都很好，但现在我怎么把这个用在3D里呢？其实很简单，为了做一个贝塞尔曲面，你需要16个控制点，( $4*4$ )，和2个变量 $t, v$ 。你要做的是计算在分量 $v$ 的沿4条平行曲线的点，再用这4个点计算在分量 $t$ 的点。计算了足够的这些点，我们可以用三角带连接他们，画出贝塞尔曲面。



恩，我认为现在已经有足够的数学背景了，看代码吧！

```
#include <math.h> // 数学库
#include <stdio.h> // 标准输入输出库
#include <stdlib.h> // 标准库

typedef struct point_3d { // 3D点的结构
    double x, y, z;
} POINT_3D;

typedef struct bpatch { // 贝塞尔面片结构
    POINT_3D anchors[4][4]; // 由4x4网格组成
    GLuint dIBPatch; // 绘制面片的显示列表名称
    GLuint texture; // 面片的纹理
} BEZIER_PATCH;

BEZIER_PATCH mybezier; // 创建一个贝塞尔曲面结构
BOOL showCPoints=TRUE; // 是否显示控制点
int divs = 7; // 细分精度，控制曲面的显示精度
```

以下是一些简单的向量数学的函数。如果你是C++爱好者你可以用一个顶点类（保证其为3D的）。

```
// 两个向量相加，p=p+q
POINT_3D pointAdd(POINT_3D p, POINT_3D q) {
    p.x += q.x;      p.y += q.y;      p.z += q.z;
    return p;
}
```

```
// 向量和标量相乘p=c*p
POINT_3D pointTimes(double c, POINT_3D p) {
    p.x *= c;      p.y *= c;      p.z *= c;
```

```

    return p;
}

// 创建一个3D向量
POINT_3D makePoint(double a, double b, double c) {
    POINT_3D p;
    p.x = a;    p.y = b;    p.z = c;
    return p;
}

```

这基本上是用C写的3维的基本函数，她用变量u和4个顶点的数组计算曲线上点。每次给u加上一定值，从0到1，我们可得一个很好的近似曲线。

求值器基于Bernstein多项式定义曲线，定义 $p(u')$ 为：

$$p(u') = B^n_i(u') R_i$$

这里 $R_i$ 为控制点

$$B^n_i(u') = [n_i] u'^i (1-u')^{n-i}$$

$$\text{且 } 0^0 = 1, [n_0] = 1$$

$$u' = (u - u_1) / (u_2 - u_1)$$

当为贝塞尔曲线时，控制点为4，相应的4个Bernstein多项式为：

$$1、 B^3_0 = (1-u)^3$$

$$2、 B^3_1 = 3u(1-u)^2$$

$$3、 B^3_2 = 3u^2(1-u)$$

$$4、 B^3_3 = u^3$$

```

// 计算贝塞尔方程的值
// 变量u的范围在0-1之间
POINT_3D Bernstein(float u, POINT_3D *p) {
    POINT_3D a, b, c, d, r;

    a = pointTimes(pow(u,3), p[0]);
    b = pointTimes(3*pow(u,2)*(1-u), p[1]);
    c = pointTimes(3*u*(1-u), p[2]);

```

```

d = pointTimes(pow((1-u),3), p[3]);

r = pointAdd(pointAdd(a, b), pointAdd(c, d));

return r;
}

```

这个函数完成共享工作，生成所有三角带，保存在display list。我们这样就不需要每贞都重新计算曲面，除了当其改变时。另外，你可能想用一个很酷的效果，用MORPHING教程改变控制点位置。这可以做一个很光滑，有机的，morphing效果，只要一点点开销（你只要改变16个点，但要从新计算）。“最后”的数组元素用来保存前一行点，（因为三角带需要两行）。而且，纹理坐标由表示百分比的u,v来计算（平面映射）。

还有一个我们没做的是计算法向量做光照。到了这一步，你基本上有2种选择。第一是找每个三角形的中心计算X，Y轴的切线，再做叉积得到垂直与两向量的向量，再归一化，得到法向量。或者（恩，这是更好的方法）你可以直接用三角形的法矢（用你最喜欢的方法计算）得到一个近似值。我喜欢后者；我认为不值得为了一点点真实感影响速度。

```

// 生成贝塞尔曲面的显示列表
GLuint genBezier(BEZIER_PATCH patch, int divs) {
    int      u = 0, v;
    float    py, px, pyold;
    GLuint   drawlist = glGenLists(1);           // 创建显示列表
    POINT_3D  temp[4];
    POINT_3D  *last = (POINT_3D*)malloc(sizeof(POINT_3D)*(divs+1)); // 更具每一条曲线的细分数，分
配相应的内存

    if (patch.dIBPatch != NULL)                  // 如果显示列表存在则删除
        glDeleteLists(patch.dIBPatch, 1);

    temp[0] = patch.anchors[0][3];                // 获得u方向的四个控制点
    temp[1] = patch.anchors[1][3];
    temp[2] = patch.anchors[2][3];
    temp[3] = patch.anchors[3][3];

    for (v=0;v<=divs;v++) {                     // 根据细分数，创建各个分割点额参数
        px = ((float)v)/((float)divs);
        // 使用Bernstein函数求的分割点的坐标
        last[v] = Bernstein(px, temp);
    }

    glNewList(drawlist, GL_COMPILE);             // 创建一个新的显示列表
    glBindTexture(GL_TEXTURE_2D, patch.texture); // 邦定纹理

    for (u=1;u<=divs;u++) {

```

```

py = ((float)u)/((float)divs);           // 计算v方向上的细分点的参数
pyold = ((float)u-1.0f)/((float)divs);    // 上一个v方向上的细分点的参数

temp[0] = Bernstein(py, patch.anchors[0]);   // 计算每个细分点v方向上贝塞尔曲面的控制点
temp[1] = Bernstein(py, patch.anchors[1]);
temp[2] = Bernstein(py, patch.anchors[2]);
temp[3] = Bernstein(py, patch.anchors[3]);

glBegin(GL_TRIANGLE_STRIP);                // 开始绘制三角形带

for (v=0;v<=divs;v++) {
    px = ((float)v)/((float)divs);        // 沿着u轴方向顺序绘制

    glTexCoord2f(pyold, px);              // 设置纹理坐标
    glVertex3d(last[v].x, last[v].y, last[v].z); // 绘制一个顶点

    last[v] = Bernstein(px, temp);       // 创建下一个顶点
    glTexCoord2f(py, px);               // 设置纹理
    glVertex3d(last[v].x, last[v].y, last[v].z); // 绘制新的顶点
}

glEnd();                                    // 结束三角形带的绘制
}

glEndList();                                // 显示列表绘制结束

free(last);                                 // 释放分配的内存
return drawlist;                            // 返回创建的显示列表
}

```

这里我们调用一个我认为有一些很酷的值的矩阵。

```

void initBezier(void) {
    mybezier.anchors[0][0] = makePoint(-0.75, -0.75, -0.50); // 设置贝塞尔曲面的控制点
    mybezier.anchors[0][1] = makePoint(-0.25, -0.75, 0.00);
    mybezier.anchors[0][2] = makePoint( 0.25, -0.75, 0.00);
    mybezier.anchors[0][3] = makePoint( 0.75, -0.75, -0.50);
    mybezier.anchors[1][0] = makePoint(-0.75, -0.25, -0.75);
    mybezier.anchors[1][1] = makePoint(-0.25, -0.25, 0.50);
    mybezier.anchors[1][2] = makePoint( 0.25, -0.25, 0.50);
    mybezier.anchors[1][3] = makePoint( 0.75, -0.25, -0.75);
    mybezier.anchors[2][0] = makePoint(-0.75, 0.25, 0.00);
    mybezier.anchors[2][1] = makePoint(-0.25, 0.25, -0.50);
}

```

```

mybezier.anchors[2][2] = makePoint( 0.25,      0.25, -0.50);
mybezier.anchors[2][3] = makePoint( 0.75,      0.25,  0.00);
mybezier.anchors[3][0] = makePoint(-0.75,     0.75, -0.50);
mybezier.anchors[3][1] = makePoint(-0.25,     0.75, -1.00);
mybezier.anchors[3][2] = makePoint( 0.25,      0.75, -1.00);
mybezier.anchors[3][3] = makePoint( 0.75,      0.75, -0.50);
mybezier.dIBPatch = NULL;                      // 默认的显示列表为0
}

```

这是一个优化的调位图的函数。可以很简单的把他们放进一个简单循环里调一组。

// 加载一个\*.bmp文件，并转化为纹理

```

BOOL LoadGLTexture(GLuint *texPntr, char* name)
{
    BOOL success = FALSE;
    AUX_RGBImageRec *TextureImage = NULL;

    glGenTextures(1, texPntr);                  // 生成纹理1

    FILE* test=NULL;
    TextureImage = NULL;

    test = fopen(name, "r");
    if (test != NULL) {
        fclose(test);
        TextureImage = auxDIBImageLoad(name);
    }

    if (TextureImage != NULL) {
        success = TRUE;

        // 邦定纹理
        glBindTexture(GL_TEXTURE_2D, *texPntr);
        glTexImage2D(GL_TEXTURE_2D, 0, 3, TextureImage->sizeX, TextureImage->sizeY, 0, GL_RGB,
        GL_UNSIGNED_BYTE, TextureImage->data);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    }

    if (TextureImage->data)
        free(TextureImage->data);
}

```

```
    return success;
}
```

仅仅加了曲面初始化在这。你每次建一个曲面时都会用这个。再一次，这里是一个用C++的好地方（贝塞尔曲面类？）。

```
int InitGL(GLvoid) // 初始化OpenGL
{
    glEnable(GL_TEXTURE_2D); // 使用2D纹理
    glShadeModel(GL_SMOOTH); // 使用平滑着色
    glClearColor(0.05f, 0.05f, 0.05f, 0.5f); // 设置黑色背景
    glClearDepth(1.0f); // 设置深度缓存
    glEnable(GL_DEPTH_TEST); // 使用深度缓存
    glDepthFunc(GL_LEQUAL); // 设置深度方程
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);

    initBezier(); // 初始化贝塞尔曲面
    LoadGLTexture(&(mybezier.texture), "./Data/NeHe.bmp"); // 载入纹理
    mybezier.dIBPatch = genBezier(mybezier, divs); // 创建显示列表

    return TRUE; // 初始化成功
}
```

首先调贝塞尔display list。再（如果边线要画）画连接控制点的线。你可以用SPACE键开关这个。

```
int DrawGLScene(GLvoid) { // 绘制场景
    int i, j;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0f, 0.0f, -4.0f); // 移入屏幕4个单位
    glRotatef(-75.0f, 1.0f, 0.0f, 0.0f);
    glRotatef(rotz, 0.0f, 0.0f, 1.0f); // 旋转一定的角度

    glCallList(mybezier.dIBPatch); // 调用显示列表，绘制贝塞尔曲面

    if (showCPoints) { // 是否绘制控制点
        glDisable(GL_TEXTURE_2D);
        glColor3f(1.0f, 0.0f, 0.0f);
        for(i=0;i<4;i++) { // 绘制水平线
            glBegin(GL_LINES);
            glVertex3f(0.0f, 0.0f, 0.0f);
            glVertex3f(0.0f, 0.0f, 1.0f);
            glEnd();
        }
    }
}
```

```

glBegin(GL_LINE_STRIP);
for(j=0;j<4;j++)
    glVertex3d(mybezier.anchors[i][j].x, mybezier.anchors[i][j].y, mybezier.anchors[i][j].z);
glEnd();
}
for(i=0;i<4;i++) { // 绘制垂直线
    glBegin(GL_LINE_STRIP);
    for(j=0;j<4;j++)
        glVertex3d(mybezier.anchors[j][i].x, mybezier.anchors[j][i].y, mybezier.anchors[j][i].z);
    glEnd();
}
glColor3f(1.0f,1.0f,1.0f);
 glEnable(GL_TEXTURE_2D);
}

return TRUE; // 成功返回
}

```

KillGLWindow()函数没有改动

CreateGLWindow( ) 函数没有改动

我在这里加了旋转曲面的代码，增加/降低分辨率，显示与否控制点连线。

```

if (keys[VK_LEFT])   rotz -= 0.8f;      // 按左键，向左旋转
if (keys[VK_RIGHT])  rotz += 0.8f; // 按右键，向右旋转
if (keys[VK_UP]) {           // 按上键，加大曲面的细分数目
    divs++;
    mybezier.dIBPatch = genBezier(mybezier, divs); // 更新贝塞尔曲面的显示列表
    keys[VK_UP] = FALSE;
}
if (keys[VK_DOWN] && divs > 1) {           // 按下键，减少曲面的细分数目
    divs--;
    mybezier.dIBPatch = genBezier(mybezier, divs); // 更新贝塞尔曲面的显示列表
    keys[VK_DOWN] = FALSE;
}
if (keys[VK_SPACE]) {           // 按空格切换控制点的可见性

```

```

showCPoints = !showCPoints;
keys[VK_SPACE] = FALSE;
}

```

恩，我希望这个教程让你了然于心而且你现在象我一样喜欢上了贝塞尔曲面。 ; - ) 如果你喜欢这个教程我会继续写一篇关于NURBS的如果有人喜欢。请EMAIL我让我知道你怎么想这篇教程。

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

#### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

#### 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

#### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。





## 第29课



Blitter 函数:

类似于DirectDraw的blit函数，过时的技术，我们有实现了它。它非常的简单，就是把一块纹理贴到另一块纹理上。

这篇文章是有Andreas Lffler所写的，它写了一份原始的教程。过了几天，Rob Fletcher发了封邮件给我，他重新改写了所有的代码，我在它的基础上把glut的框架变换为Win32的框架。

现在让我们开始吧！

下面是一个保存图像数据的结构

```
typedef struct Texture_Image
{
    int width;                      // 宽
    int height;                     // 高
    int format;                     // 像素格式
```

```
unsigned char *data; // 纹理数据
} TEXTURE_IMAGE;
```

接下来定义了两个指向这个结构的指针

```
typedef TEXTURE_IMAGE *P_TEXTURE_IMAGE;
P_TEXTURE_IMAGE t1; // 指向保存图像结构的指针
P_TEXTURE_IMAGE t2; // 指向保存图像结构的指针
```

下面的函数为w\*h的图像分配内存

```
P_TEXTURE_IMAGE AllocateTextureBuffer( GLint w, GLint h, GLint f )
{
    P_TEXTURE_IMAGE ti=NULL;
    unsigned char *c=NULL;

    ti = (P_TEXTURE_IMAGE)malloc(sizeof(TEXTURE_IMAGE)); // 分配图像结构
    内存

    if( ti != NULL ) {
        ti->width = w; // 设置宽度
        ti->height = h; // 设置高度
        ti->format = f; // 设置格式
        // 分配w*h*f个字节
        c = (unsigned char *)malloc( w * h * f );
        if ( c != NULL ) {
            ti->data = c;
        }
        else {
            MessageBox(NULL,"内存不足","分配图像内存错误",MB_OK | MB_ICONINFORMATION);
            return NULL;
        }
    }
}
```

```

    }

else
{
    MessageBox(NULL,"内存不足","分配图像结构内存错误",MB_OK | MB_ICONINFORMATION);
    return NULL;
}
return ti;                                // 返回指向图像数据的指针
}

```

下面的函数释放分配的内存

```

// 释放图像内存
void DeallocateTexture( P_TEXTURE_IMAGE t )
{
    if(t)
    {
        if(t->data)
        {
            free(t->data);                // 释放图像内存
        }

        free(t);                        // 释放图像结构内存
    }
}

```

下面我们来读取\*.raw的文件，这个函数有两个参数，一个为文件名，另一个为保存文件的图像结构指针。

```

// 读取*.RAW文件，并把图像文件上下翻转一符合OpenGL的使用格式。
int ReadTextureData ( char *filename, P_TEXTURE_IMAGE buffer)
{
    FILE *f;
    int i,j,k,done=0;

```

```

int stride = buffer->width * buffer->format;           // 记录每一行的宽度 , 以字节为单位
unsigned char *p = NULL;

f = fopen(filename, "rb");                                // 打开文件
if( f != NULL )                                         // 如果文件存在
{

```

如果文件存在 , 我们通过一个循环读取我们的纹理 , 我们从图像的最下面一行 , 一行一行的读取图像。

```

for( i = buffer->height-1; i >= 0 ; i-- )           // 循环所有的行 , 从最下面以行开始 , 一行一行的读取
{
    p = buffer->data + (i * stride );
    for ( j = 0; j < buffer->width ; j++ )          // 读取每一行的数据
    {

```

下面的循环读取每一像素的数据 , 并把alpha设为255

```

for ( k = 0 ; k < buffer->format-1 ; k++, p++, done++ )
{
    *p = fgetc(f);                                // 读取一个字节
}
*p = 255; p++;                                       // 把255存储在alpha通道中
}
fclose(f);                                           // 关闭文件
}

```

如果出现错误 , 弹出一个提示框

```

else
{
    MessageBox(NULL,"不能打开文件","图像错误",MB_OK | MB_ICONINFORMATION);
}
return done; // 返回读取的字节数
}

```

下面的代码创建一个2D纹理，和前面课程介绍的方法相同

```

void BuildTexture (P_TEXTURE_IMAGE tex)
{
    glGenTextures(1, &texture[0]);
    glBindTexture(GL_TEXTURE_2D, texture[0]);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
    gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB, tex->width, tex->height, GL_RGBA,
GL_UNSIGNED_BYTE, tex->data);
}

```

现在到了blitter函数的地方了，他运行你把一个图像的任意部分复制到另一个图像的任意部分，并混合。

src为原图像

dst为目标图像

src\_xstart,src\_ystart为要复制的部分在原图像中的位置

src\_width,src\_height为要复制的部分的宽度和高度

dst\_xstart,dst\_ystart为复制到目标图像时的起始位置

上面的意思是把原图像中的(src\_xstart,src\_ystart)-(src\_width,src\_height)复制到目标图像中(dst\_xstart,dst\_ystart)-(src\_width,src\_height)

blend设置是否启用混合，0为不启用，1为启用

alpha设置源图像中颜色在混合时所占的百分比

```

void Blit( P_TEXTURE_IMAGE src, P_TEXTURE_IMAGE dst, int src_xstart, int src_ystart, int src_width,
int src_height,
        int dst_xstart, int dst_ystart, int blend, int alpha)
{

```

```

int i,j,k;
unsigned char *s, *d;

// 捉断alpha的值
if( alpha > 255 ) alpha = 255;
if( alpha < 0 ) alpha = 0;

// 判断是否启用混合
if( blend < 0 ) blend = 0;
if( blend > 1 ) blend = 1;

d = dst->data + (dst_ystart * dst->width * dst->format);           // 要复制的像素在目标图像数
据中的开始位置
s = src->data + (src_ystart * src->width * src->format);           // 要复制的像素在源图像数据中
的开始位置

for (i = 0 ; i < src_height ; i++)
{
    // 循环每一行

    s = s + (src_xstart * src->format);                                // 移动到下一个像素
    d = d + (dst_xstart * dst->format);
    for (j = 0 ; j < src_width ; j++)                                     // 循环复制一行
    {
        for( k = 0 ; k < src->format ; k++, d++, s++)
            // 复制每一个字节

        if (blend)                                              // 如果启用了混合
            *d = ( (*s * alpha) + (*d * (255-alpha)) ) >> 8;    // 根据混合复制颜色
        else
            *d = *s;                                            // 否则直接复制
    }
    d = d + (dst->width - (src_width + dst_xstart))*dst->format;      // 移动到下一行
    s = s + (src->width - (src_width + src_xstart))*src->format;
}
}

```

初始化代码基本不变，我们使用新的函数，加载\*.raw纹理。并把纹理t2的一部分blit到t1中混合，接着按常规的方法设置2D纹理。

```

int InitGL(GLvoid)
{
    t1 = AllocateTextureBuffer( 256, 256, 4 ); // 为图像t1分配内存
    if (ReadTextureData("Data/Monitor.raw",t1)==0) // 读取图像数据
    {
        MessageBox(NULL,"不能读取 'Monitor.raw' 文件","读取错误",MB_OK | MB_ICONINFORMATION);
        return FALSE;
    }

    t2 = AllocateTextureBuffer( 256, 256, 4 ); // 为图像t2分配内存
    if (ReadTextureData("Data/GL.raw",t2)==0) // 读取图像数据
    {
        MessageBox(NULL,"不能读取 'GL.raw' 文件","读取错误 ",MB_OK | MB_ICONINFORMATION);
        return FALSE;
    }
}

```

把图像t2的 ( 127 , 127 ) - ( 256 , 256 ) 部分和图像t1的 ( 64 , 64 , 196 , 196 ) 部分混合

// 把图像t2的 ( 127 , 127 ) - ( 256 , 256 ) 部分和图像t1的 ( 64 , 64 , 196 , 196 ) 部分混合  
 Blit(t2,t1,127,127,128,128,64,64,1,127);

下面的代码和前面一样，释放分配的空间，创建纹理

```

BuildTexture (t1); // 把t1图像加载为纹理

DeallocateTexture( t1 ); // 释放图像数据
DeallocateTexture( t2 );

glEnable(GL_TEXTURE_2D); // 使用2D纹理

glShadeModel(GL_SMOOTH); // 使用光滑着色
glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // 设置背景色为黑色

```

```

glClearDepth(1.0);                                // 设置深度缓存清楚值为1
glEnable(GL_DEPTH_TEST);                          // 使用深度缓存
glDepthFunc(GL_LESS);                            // 设置深度测试函数

return TRUE;
}

```

下面的代码绘制一个盒子

```

GLvoid DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);          // 清楚颜色缓存和
深度缓存
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-5.0f);

    glRotatef(xrot,1.0f,0.0f,0.0f);
    glRotatef(yrot,0.0f,1.0f,0.0f);
    glRotatef(zrot,0.0f,0.0f,1.0f);

    glBindTexture(GL_TEXTURE_2D, texture[0]);

    glBegin(GL_QUADS);
        // 前面
        glNormal3f( 0.0f, 0.0f, 1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
        // 后面
        glNormal3f( 0.0f, 0.0f, -1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
        // 上面
        glNormal3f( 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);

```

```
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
// 下面
glNormal3f( 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);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
// 右面
glNormal3f( 1.0f, 0.0f, 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);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
// 左面
glNormal3f(-1.0f, 0.0f, 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);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glEnd();
```

```
xrot+=0.3f;
yrot+=0.2f;
zrot+=0.4f;
```

```
return TRUE; //一切OK
```

```
}
```

KillGLWindow() 函数没有变化

CreateGLWindow函数没有变化

WinMain() 没有变化

好了，现你可以很轻松的绘制很多混合效果。如果你有什么好的建议，请告诉我。



### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

### 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望

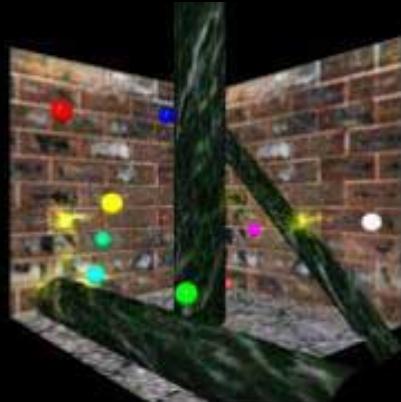
||我能带给她幸福。

< 第28课

第30课 >



## 第30课



碰撞检测:

这是一课激动的教程 , 你也许等待它多时了。你将学会碰撞剪裁 , 物理模拟太多的东西 , 慢慢期待吧。

碰撞检测和物理模拟(作者:Dimitrios Christopoulos (christop@fhw.gr))

### 碰撞检测

这是一个我遇到的最困难的题目,因为它没有一个简单的解决办法.对于每一个程序都有一种检测碰撞的方法.当然这里有一种蛮力,它适用于各种不同的应用,当它非常的费时.

我们将讲述一种算法,它非常的快,简单并易于扩展.下面我们来看看这个算法包含的内容:

#### 1) 碰撞检测

移动的球-平面  
移动的球-圆柱  
移动的球-移动的球

#### 2) 基于物理的建模

碰撞表示  
应用重力加速度

#### 3) 特殊效果

爆炸的表示 , 利用互交叉的公告板形式  
声音使用Windows声音库

#### 4) 关于代码

代码被分为以下5个部分

Lesson30.cpp : 主程序代码  
 Image.cpp, Image.h : 加载图像  
 Tmatrix.cpp, Tmatrix.h : 矩阵  
 Tray.cpp, Tray.h : 射线  
 Tvector.cpp, Tvector.h : 向量

### 1) 碰撞检测

我们使用射线来完成相关的算法，它的定义为：

射线上的点 = 射线的原点 +  $t * \text{射线的方向}$

$t$  用来描述它距离原点的位置，它的范围是[0, 无限远).

现在我们可以使用射线来计算它和平面以及圆柱的交点了。

射线和平面的碰撞检测：

平面被描述为：

$$\mathbf{X}_n \cdot \mathbf{X} = d$$

$\mathbf{X}_n$  是平面的法线.

$\mathbf{X}$  是平面上的一个点.

$d$  是平面到原点的距离.

现在我们得到射线和平面的两个方程：

$$\text{PointOnRay} = \text{Raystart} + t * \text{Raydirection}$$

$$\mathbf{X}_n \cdot \mathbf{X} = d$$

如果他们相交，则上诉方程组有解，如下所示：

$$\mathbf{X}_n \cdot \text{PointOnRay} = d$$

$$(\mathbf{X}_n \cdot \text{Raystart}) + t * (\mathbf{X}_n \cdot \text{Raydirection}) = d$$

解得  $t$ :

$$t = (d - \mathbf{X}_n \cdot \text{Raystart}) / (\mathbf{X}_n \cdot \text{Raydirection})$$

$t$  代表原点到与平面相交点的参数, 把  $t$  带回原方程我们会得到与平面的碰撞点. 如果  $\mathbf{X}_n \cdot \text{Raydirection} = 0$ 。 则说明它与平面平行，则将不产生碰撞。如果  $t$  为负值，则说明交点在射线的相反方向，也不会产生碰撞。

---

```
//判断是否和平面相交，是则返回1，否则返回0
int TestIntersectionPlane(const Plane& plane,const TVector& position,const TVector& direction,double& lambda,TVector& pNormal)
```

{

```
double DotProduct=direction.dot(plane._Normal);
double l2;

//判断是否平行于平面
if ((DotProduct<ZERO)&&(DotProduct>-ZERO))
return 0;

l2=(plane._Normal.dot(plane._Position-position))/DotProduct;

if (l2<-ZERO)
return 0;

pNormal=plane._Normal;
lamda=l2;
return 1;
}
```

### 射线-圆柱的碰撞检测

计算射线和圆柱方程组得解。

```
int TestIntersionCylinder(const Cylinder& cylinder,const TVector& position,const TVector& direction, double& lamda,
TVector& pNormal,TVector& newposition)
```

### 球-球之间的碰撞检测

球被表示为中心和它的半径，决定两个球是否相交就是求出它们之间的距离是否小于它们的直径。

在处理两个移动的球是否相交时，有一个bug就是，当它们的移动速度太快，回出现它们相交，但在相邻的两步检测不出它们是否相交的情况，如下图所示：

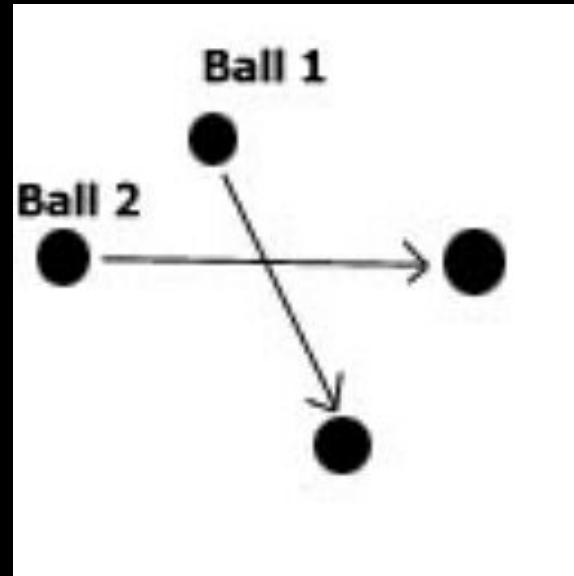


图 1

有一个替代的办法就是细分相邻的时间片断，如果在这之间发生了碰撞，则确定有效。我们把这个细分时间段设置为3，代码如下：

```
//判断球和球是否相交，是则返回1，否则返回0
int FindBallCol(TVector& point, double& TimePoint, double Time2, int& BallNr1, int& BallNr2)
{
    TVector RelativeV;
    TRay rays;
    double MyTime=0.0, Add=Time2/150.0, Timedummy=10000, Timedummy2=-1;
    TVector pos;
    //判断球和球是否相交
    for (int i=0;i<NrOfBalls-1;i++)
    {
        for (int j=i+1;j<NrOfBalls;j++)
        {
            RelativeV=ArrayVel[i]-ArrayVel[j];
            rays=TRay(OldPos[i],TVector::unit(RelativeV));
            MyTime=0.0;

            if ((rays.dist(OldPos[j])) > 40) continue;

            while (MyTime<Time2)
            {
                MyTime+=Add;
                pos=OldPos[i]+RelativeV*MyTime;
                if (pos.dist(OldPos[j])<=40) {
                    point=pos;
                    if (Timedummy>(MyTime-Add)) Timedummy=MyTime-Add;
                    BallNr1=i;
                    BallNr2=j;
                    break;
                }
            }
        }
    }
}
```

```

}
}

}

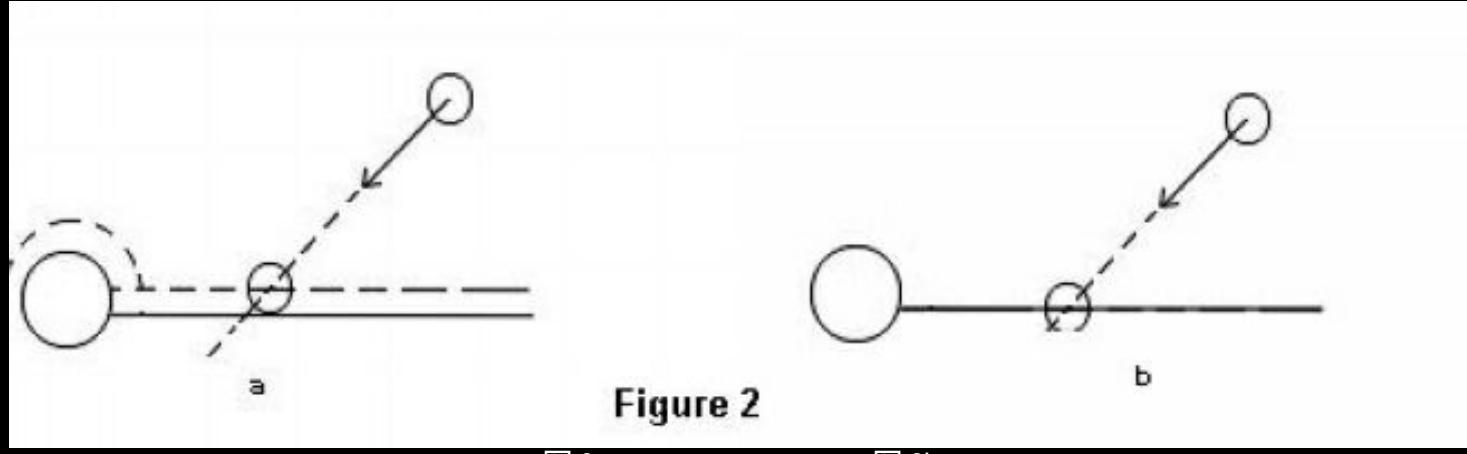
if (Timedummy!=10000) { TimePoint=Timedummy;
return 1;
}

return 0;
}

```

### 怎样应用我们的知识

现在我们已经可以决定射线和平面/圆柱的交点了,如下图所示:



当我们找到了碰撞位置后,下一步我们需要知道它是否发生在当前这一步中.如果距离碰撞点的位置小于这一步球体运动的间隔,则碰撞发生.我们使用如下的方程计算运动到碰撞时所需的时间:

$$T_c = D_{sc} * T / D_{st}$$

接着我们知道碰撞点位置,如下面公式所示:

$$\text{Collision point} = \text{Start} + \text{Velocity} * T_c$$

### 2) 基于物理的模拟

#### 碰撞反应

为了计算对于一个静止物体的碰撞,我们需要知道以下信息:碰撞点,碰撞法线,碰撞时间.

它是基于以下物理规律的,碰撞的入射角等于反射角.如下图所示:

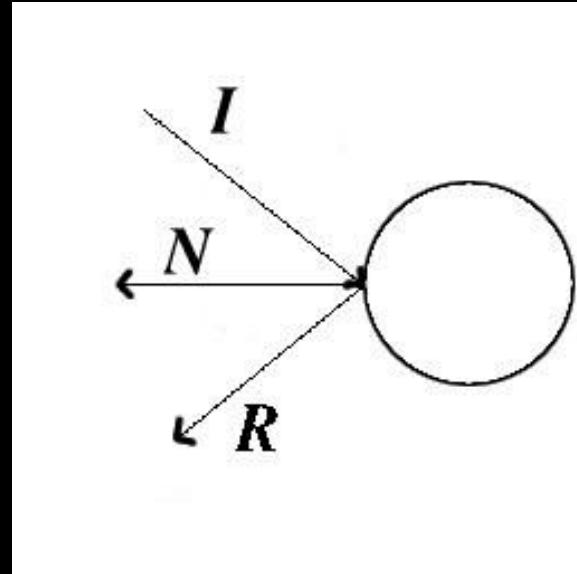


图 3

R 为反射方向

I 为入射方向

N 为法线方向

反射方向有以下公式计算：

$$R = 2 * (-I \cdot N) * N + I$$

```

rt2=ArrayVel[BallNr].mag();           // 返回速度向量的模
ArrayVel[BallNr].unit();              // 归一化速度向量

// 计算反射向量
ArrayVel[BallNr]=TVector::unit( (normal*(2*normal.dot(-ArrayVel[BallNr]))) + ArrayVel[BallNr] );
ArrayVel[BallNr]=ArrayVel[BallNr]*rt2;

```

### 球体之间的碰撞

由于它很复杂，我们用下图来说明这个原理。

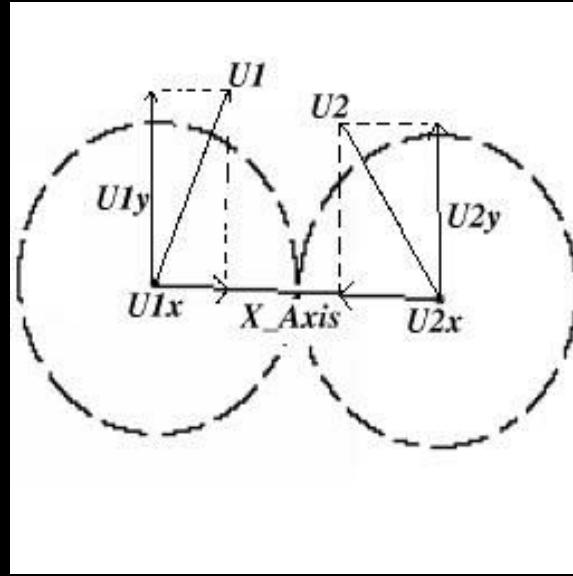


图 4

$U_1$ 和 $U_2$ 为速度向量，我们用 $X\_Axis$ 表示两个球中心连线的轴， $U_{1X}$ 和 $U_{2X}$ 为 $U_1$ 和 $U_2$ 在这个轴上的分量。 $U_{1y}$ 和 $U_{2y}$ 为垂直于 $X\_Axis$ 轴的分量。 $M_1$ 和 $M_2$ 为两个球体的分量。 $V_1$ 和 $V_2$ 为碰撞后的速度， $V_{1x}, V_{1y}, V_{2x}, V_{2y}$ 为他们的分量。

在我们的例子里，所有球的质量都相等，解得方程为，在垂直轴上的速度不变，在 $X\_Axis$ 轴上互相交换速度。代码如下：

```

TVector pb1,pb2,xaxis,U1x,U1y,U2x,U2y,V1x,V1y,V2x,V2y;
double a,b;
pb1=OldPos[BallColNr1]+ArrayVel[BallColNr1]*BallTime;           // 球1的位置
pb2=OldPos[BallColNr2]+ArrayVel[BallColNr2]*BallTime;           // 球2的位置
xaxis=(pb2-pb1).unit();                                         // X-Axis轴
a=xaxis.dot(ArrayVel[BallColNr1]);                                // X_Axis投影系数
U1x=xaxis*a;                                                    // 计算在X_Axis轴上的速度
U1y=ArrayVel[BallColNr1]-U1x; // 计算在垂直轴上的速度
xaxis=(pb1-pb2).unit();
b=xaxis.dot(ArrayVel[BallColNr2]);
U2x=xaxis*b;
U2y=ArrayVel[BallColNr2]-U2x;
V1x=(U1x+U2x-(U1x-U2x))*0.5;                                // 计算新的速度
V2x=(U1x+U2x-(U2x-U1x))*0.5;
V1y=U1y;
V2y=U2y;
for (j=0;j<NrOfBalls;j++)                                     // 更新所有球的位置
    ArrayPos[j]=OldPos[j]+ArrayVel[j]*BallTime;
    ArrayVel[BallColNr1]=V1x+V1y;                                // 设置新的速度
    ArrayVel[BallColNr2]=V2x+V2y;

```

## 万有引力的模拟

我们使用欧拉方程来模拟万有引力，如下所示：

Velocity\_New = Velocity\_Old + Acceleration\*TimeStep  
 Position\_New = Position\_Old + Velocity\_New\*TimeStep

在每次模拟中，我们用上面公式计算的速度取代旧的速度

### 3) 特殊效果

#### 爆炸

最好的表示爆炸效果的就是使用两个互相垂直的平面，并使用alpha混合在窗口中显示它们。接着让alpha变为0，设定爆炸效果不可见。代码如下所示：

```
// 渲染/混合爆炸效果
glEnable(GL_BLEND);           // 使用混合
glDepthMask(GL_FALSE);        // 禁用深度缓存
glBindTexture(GL_TEXTURE_2D, texture[1]); // 设置纹理
for(i=0; i<20; i++)          // 渲染20个爆炸效果
{
    if(ExplosionArray[i]._Alpha>=0)
    {
        glPushMatrix();
        ExplosionArray[i]._Alpha-=0.01f; // 设置alpha
        ExplosionArray[i]._Scale+=0.03f; // 设置缩放
        // 设置颜色
        glColor4f(1,1,0,ExplosionArray[i]._Alpha);
        glScalef(ExplosionArray[i]._Scale,ExplosionArray[i]._Scale,ExplosionArray[i]._Scale);
        // 设置位置
        glTranslatef((float)ExplosionArray[i]._Position.X()/ExplosionArray[i]._Scale,
                     (float)ExplosionArray[i]._Position.Y()/ExplosionArray[i]._Scale,
                     (float)ExplosionArray[i]._Position.Z()/ExplosionArray[i]._Scale);
        glCallList(dlist); // 调用显示列表绘制爆炸效果
        glPopMatrix();
    }
}
```

## 声音

在Windows下我们简单的调用PlaySound()函数播放声音。

### 4) 代码的流程

如果你成功的读完了理论部分，在你开始运行程序并播放声音以前。我们将用伪代码向你介绍一些整个流程，以便你能成功的看懂代码。

While (Timestep!=0)

```

{
    对每一个球
    {
        计算最近的与平面碰撞的位置;
        计算最近的与圆柱碰撞的位置;
        如果碰撞发生，则保存并替换最近的碰撞点;
    }
    检测各个球之间的碰撞;
    如果碰撞发生，则保存并替换最近的碰撞点;

    If (碰撞发生)
    {
        移动所有的球道碰撞点的时间;
        (We already have computed the point, normal and collision time.)
        计算碰撞后的效果;
        Timestep-=CollisionTime;
    }
    else
        移动所有的球体一步
}

```

下面是对上面伪代码的实现：

```

//模拟函数，计算碰撞检测和物理模拟
void idle()
{
    double rt,rt2,rt4,lamda=10000;
    TVector norm,uveloc;
    TVector normal,point,time;
    double RestTime,BallTime;
    TVector Pos2;
    int BallNr=0,dummy=0,BallColNr1,BallColNr2;
    TVector Nc;

//如果没有锁定到球上，旋转摄像机
if (!hook_toball1)
{
    camera_rotation+=0.1f;
    if (camera_rotation>360)
        camera_rotation=0;
}

RestTime=Time;
lamda=1000;

//计算重力加速度
for (int j=0;j<NrOfBalls;j++)

```

```
ArrayVel[j]+=accel*RestTime;
```

```
//如果在一步的模拟时间内(如果来不及计算，则跳过几步)
```

```
while (RestTime>ZERO)
```

```
{
```

```
lamda=10000;
```

```
//对于每个球，找到它们最近的碰撞点
```

```
for (int i=0;i<NrOfBalls;i++)
```

```
{
```

```
//计算新的位置和移动的距离
```

```
OldPos[i]=ArrayPos[i];
```

```
TVector::unit(ArrayVel[i],uveloc);
```

```
ArrayPos[i]=ArrayPos[i]+ArrayVel[i]*RestTime;
```

```
rt2=OldPos[i].dist(ArrayPos[i]);
```

```
//测试是否和墙面碰撞
```

```
if (TestIntersionPlane(pl1,OldPos[i],uveloc,rt,norm))
```

```
{
```

```
//计算碰撞的时间
```

```
rt4=rt*RestTime/rt2;
```

```
//如果小于当前保存的碰撞时间，则更新它
```

```
if (rt4<=lamda)
```

```
{
```

```
if (rt4<=RestTime+ZERO)
```

```
if (! ((rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )
```

```
{
```

```
normal=norm;
```

```
point=OldPos[i]+uveloc*rt;
```

```
lamda=rt4;
```

```
BallNr=i;
```

```
}
```

```
}
```

```
}
```

```
if (TestIntersionPlane(pl2,OldPos[i],uveloc,rt,norm))
```

```
{
```

```
rt4=rt*RestTime/rt2;
```

```
if (rt4<=lamda)
```

```
{
```

```
if (rt4<=RestTime+ZERO)
```

```
if (! ((rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )
```

```
{
```

```
normal=norm;
```

```
point=OldPos[i]+uveloc*rt;
```

```
lamda=rt4;
```

```
BallNr=i;
```

```
dummy=1;
```

```
}
```

```
}
```

}

```
if (TestIntersionPlane(pl3,OldPos[i],uveloc,rt,norm))  
{  
rt4=rt*RestTime/rt2;
```

```
if (rt4<=lamda)  
{  
if (rt4<=RestTime+ZERO)  
if (!(rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )  
{  
normal=norm;  
point=OldPos[i]+uveloc*rt;  
lamda=rt4;  
BallNr=i;  
}  
}  
}
```

```
if (TestIntersionPlane(pl4,OldPos[i],uveloc,rt,norm))  
{  
rt4=rt*RestTime/rt2;
```

```
if (rt4<=lamda)  
{  
if (rt4<=RestTime+ZERO)  
if (!(rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )  
{  
normal=norm;  
point=OldPos[i]+uveloc*rt;  
lamda=rt4;  
BallNr=i;  
}  
}  
}
```

```
if (TestIntersionPlane(pl5,OldPos[i],uveloc,rt,norm))  
{  
rt4=rt*RestTime/rt2;
```

```
if (rt4<=lamda)  
{  
if (rt4<=RestTime+ZERO)  
if (!(rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )  
{  
normal=norm;  
point=OldPos[i]+uveloc*rt;  
lamda=rt4;  
BallNr=i;  
}
```

}

}

```
//测试是否与三个圆柱相碰
if (TestIntersionCylinder(cyl1,OldPos[i],uveloc,rt,norm,Nc))
{
rt4=rt*RestTime/rt2;

if (rt4<=lamda)
{
if (rt4<=RestTime+ZERO)
if (! ((rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )
{
normal=norm;
point=Nc;
lamda=rt4;
BallNr=i;
}
}

if (TestIntersionCylinder(cyl2,OldPos[i],uveloc,rt,norm,Nc))
{
rt4=rt*RestTime/rt2;

if (rt4<=lamda)
{
if (rt4<=RestTime+ZERO)
if (! ((rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )
{
normal=norm;
point=Nc;
lamda=rt4;
BallNr=i;
}

if (TestIntersionCylinder(cyl3,OldPos[i],uveloc,rt,norm,Nc))
{
rt4=rt*RestTime/rt2;

if (rt4<=lamda)
{
if (rt4<=RestTime+ZERO)
if (! ((rt<=ZERO)&&(uveloc.dot(norm)>ZERO)) )
{
normal=norm;
point=Nc;
lamda=rt4;
BallNr=i;
}
```

{  
}  
}  
}

//计算每个球之间的碰撞，如果碰撞时间小于与上面的碰撞，则替换它们  
if (FindBallCol(Pos2,BallTime,RestTime,BallColNr1,BallColNr2))  
{  
if (sounds)  
PlaySound("Data/Explode.wav",NULL,SND\_FILENAME|SND\_ASYNC);  
  
if ( (lamda==10000) || (lamda>BallTime) )  
{  
RestTime=RestTime-BallTime;  
  
TVector pb1,pb2,xaxis,U1x,U1y,U2x,U2y,V1x,V1y,V2x,V2y;  
double a,b;  
  
pb1=OldPos[BallColNr1]+ArrayVel[BallColNr1]\*BallTime;  
pb2=OldPos[BallColNr2]+ArrayVel[BallColNr2]\*BallTime;  
xaxis=(pb2-pb1).unit();  
  
a=xaxis.dot(ArrayVel[BallColNr1]);  
U1x=xaxis\*a;  
U1y=ArrayVel[BallColNr1]-U1x;  
  
xaxis=(pb1-pb2).unit();  
b=xaxis.dot(ArrayVel[BallColNr2]);  
U2x=xaxis\*b;  
U2y=ArrayVel[BallColNr2]-U2x;  
  
V1x=(U1x+U2x-(U1x-U2x))\*0.5;  
V2x=(U1x+U2x-(U2x-U1x))\*0.5;  
V1y=U1y;  
V2y=U2y;  
  
for (j=0;j<NrOfBalls;j++)  
ArrayPos[j]=OldPos[j]+ArrayVel[j]\*BallTime;  
  
ArrayVel[BallColNr1]=V1x+V1y;  
ArrayVel[BallColNr2]=V2x+V2y;  
  
//Update explosion array  
for(j=0;j<20;j++)  
{  
if (ExplosionArray[j].\_Alpha<=0)  
{  
ExplosionArray[j].\_Alpha=1;  
ExplosionArray[j].\_Position=ArrayPos[BallColNr1];  
ExplosionArray[j].\_Scale=1;

```
break;
}
}

continue;
}
}

//最后的测试，替换下次碰撞的时间，并更新爆炸效果的数组
if (lamda!=10000)
{
RestTime-=lamda;

for (j=0;j<NrOfBalls;j++)
ArrayPos[j]=OldPos[j]+ArrayVel[j]*lamda;

rt2=ArrayVel[BallNr].mag();
ArrayVel[BallNr].unit();
ArrayVel[BallNr]=TVector::unit( (normal*(2*normal.dot(-ArrayVel[BallNr]))) + ArrayVel[BallNr] );
ArrayVel[BallNr]=ArrayVel[BallNr]*rt2;

for(j=0;j<20;j++)
{
if (ExplosionArray[j]._Alpha<=0)
{
ExplosionArray[j]._Alpha=1;
ExplosionArray[j]._Position=point;
ExplosionArray[j]._Scale=1;
break;
}
}
}
else
RestTime=0;

}
}
```

你可以从源代码得到全部的信息,我尽了最大的努力来解释每一行代码,一旦碰撞的原理知道了,代码是非常简单的.

就像我开头所说的,碰撞检测这个题目是非常难得,你已经学会了很多新的知识,并能够用它创建出非常棒的演示.但在这个课题,你还有很多需要学习,既然你已经开始了,其它的原理和模型就非常容易了.

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

#### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

#### 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

#### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

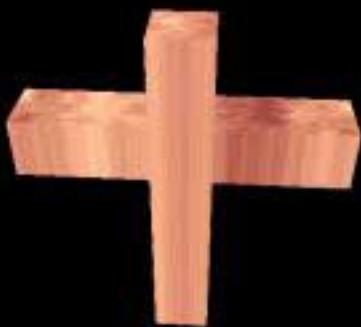
感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。

< 第29课

第31课 >



## 第31课



### 模型加载:

你知道大名鼎鼎的Milkshape3D建模软件么，我们将加载它的模型，当然你可以加载任何你认为不错的模型。

这篇渲染模型的文章是由Brett Porter所写的。

这篇教程的代码是从PortaLib3D中提取出来的，PortaLib3D是一个可以读取3D文件实用库。

这篇教程的代码是以第六课为基础的，我们只讨论改变的部分。

这课中使用的模型是从Milkshape3D中提取出来的，Milkshape3D是一个非常好的建模软件，它包含了自己的文件格式，所以你能很容易去分析和理解。

但是文件格式并不能使你加载一个模型，你必须自己定义一个结构去保存数据，接着把数据读入那个结构，我们将告诉你如何定义这样一个结构。

模型的定义在model.h中，好吧我们开始吧：

```
// 顶点结构
```

```
struct Vertex
{
    char m_boneID; // 顶点所在的骨骼
    float m_location[3];
};

// 顶点的个数和数据
int m_numVertices;
Vertex *m_pVertices;
```

在这一课你，你可以忽略m\_boneID，我们将在以后的教程中介绍骨骼动画。m\_location 定义顶点的位置。

下面是三角形结构

```
// 三角形结构
struct Triangle
{
    float m_vertexNormals[3][3];
    float m_s[3], m_t[3];
    int m_vertexIndices[3];
};

// 使用的三角形
int m_numTriangles;
Triangle *m_pTriangles;
```

3个顶点构成一个三角形，m\_vertexIndices保存了三个顶点的索引。m\_s 和 m\_t储存了三个顶点的纹理坐标。m\_vertexNormals保存了三个顶点的法线。

下面我们定义网格结构

```
// 网格结构
struct Mesh
{
    int m_materialIndex;
```

```

int m_numTriangles;
int *m_pTriangleIndices;
};

// 使用的网格
int m_numMeshes;
Mesh *m_pMeshes;

```

m\_pTriangleIndices指向包含在网格中三角形的数据，它是动态分配的。m\_materialIndex指向了这个网格所用的材质。

```

// 材质属性
struct Material
{
    float m_ambient[4], m_diffuse[4], m_specular[4], m_emissive[4];
    float m_shininess;
    GLuint m_texture;
    char *m_pTextureFilename;
};

// 使用的纹理
int m_numMaterials;
Material *m_pMaterials;

```

这里我们使用与OpenGL中相对的材质。

下面的代码用来载入模型，我们通过重载loadModelData函数来实现它。

我们创建了一个新类MilkshapeModel,它是从Model继承而来的。

```

bool MilkshapeModel::loadModelData( const char *filename )
{
    ifstream inputFile( filename, ios::in | ios::binary | ios::nocreate );
    if ( inputFile.fail() )
        return false; // 不能打开文件，返回失败
}

```

以二进制的方式打开文件，如果失败则返回

```
inputFile.seekg( 0, ios::end );
long fileSize = inputFile.tellg();
inputFile.seekg( 0, ios::beg );
```

返回文件大小

```
byte *pBuffer = new byte[fileSize];
inputFile.read( pBuffer, fileSize );
inputFile.close();
```

分配一个内存，载入文件，并关闭文件

```
const byte *pPtr = pBuffer;
MS3DHeader *pHeader = ( MS3DHeader* )pPtr;
pPtr += sizeof( MS3DHeader );

if ( strncmp( pHeader->m_ID, "MS3D000000", 10 ) != 0 )
    return false; // 如果不是一个有效的MS3D文件则返回

if ( pHeader->m_version < 3 || pHeader->m_version > 4 )
    return false; // 如果不能支持这种版本的文件，则返回失败
```

上面的文件读取文件头

```

int nVertices = *( word* )pPtr;
m_numVertices = nVertices;
m_pVertices = new Vertex[nVertices];
pPtr += sizeof( word );

int i;
for ( i = 0; i < nVertices; i++ )
{
    MS3DVertex *pVertex = ( MS3DVertex* )pPtr;
    m_pVertices[i].m_boneID = pVertex->m_boneID;
    memcpy( m_pVertices[i].m_location, pVertex->m_vertex, sizeof( float )*3 );
    pPtr += sizeof( MS3DVertex );
}

```

上面的代码读取顶点数据

```

int nTriangles = *( word* )pPtr;
m_numTriangles = nTriangles;
m_pTriangles = new Triangle[nTriangles];
pPtr += sizeof( word );

for ( i = 0; i < nTriangles; i++ )
{
    MS3DTriangle *pTriangle = ( MS3DTriangle* )pPtr;
    int vertexIndices[3] = { pTriangle->m_vertexIndices[0], pTriangle->m_vertexIndices[1], pTriangle-
>m_vertexIndices[2] };
    float t[3] = { 1.0f-pTriangle->m_t[0], 1.0f-pTriangle->m_t[1], 1.0f-pTriangle->m_t[2] };
    memcpy( m_pTriangles[i].m_vertexNormals, pTriangle->m_vertexNormals, sizeof( float )*3*3 );
    memcpy( m_pTriangles[i].m_s, pTriangle->m_s, sizeof( float )*3 );
    memcpy( m_pTriangles[i].m_t, t, sizeof( float )*3 );
    memcpy( m_pTriangles[i].m_vertexIndices, vertexIndices, sizeof( int )*3 );
    pPtr += sizeof( MS3DTriangle );
}

```

上面的代码用来读取三角形信息 , 因为MS3D使用窗口坐标系而OpenGL使用笛卡儿坐标系 , 所以需要反转每个顶点Y方向的纹理坐标

```
int nGroups = *( word* )pPtr;
m_numMeshes = nGroups;
m_pMeshes = new Mesh[nGroups];
pPtr += sizeof( word );
for ( i = 0; i < nGroups; i++ )
{
    pPtr += sizeof( byte );
    pPtr += 32;

    word nTriangles = *( word* )pPtr;
    pPtr += sizeof( word );
    int *pTriangleIndices = new int[nTriangles];
    for ( int j = 0; j < nTriangles; j++ )
    {
        pTriangleIndices[j] = *( word* )pPtr;
        pPtr += sizeof( word );
    }

    char materialIndex = *( char* )pPtr;
    pPtr += sizeof( char );

    m_pMeshes[i].m_materialIndex = materialIndex;
    m_pMeshes[i].m_numTriangles = nTriangles;
    m_pMeshes[i].m_pTriangleIndices = pTriangleIndices;
}
```

上面的代码填充网格结构

```
int nMaterials = *( word* )pPtr;
m_numMaterials = nMaterials;
m_pMaterials = new Material[nMaterials];
pPtr += sizeof( word );
for ( i = 0; i < nMaterials; i++ )
{
    MS3DMaterial *pMaterial = ( MS3DMaterial* )pPtr;
    memcpy( m_pMaterials[i].m_ambient, pMaterial->m_ambient, sizeof( float )*4 );
```

```

    memcpy( m_pMaterials[i].m_diffuse, pMaterial->m_diffuse, sizeof( float )*4 );
    memcpy( m_pMaterials[i].m_specular, pMaterial->m_specular, sizeof( float )*4 );
    memcpy( m_pMaterials[i].m_emissive, pMaterial->m_emissive, sizeof( float )*4 );
    m_pMaterials[i].m_shininess = pMaterial->m_shininess;
    m_pMaterials[i].m_pTextureFilename = new char[strlen( pMaterial->m_texture )+1];
    strcpy( m_pMaterials[i].m_pTextureFilename, pMaterial->m_texture );
    pPtr += sizeof( MS3DMaterial );
}

reloadTextures();

```

上面的代码加载纹理数据

```

delete[] pBuffer;

return true;
}

```

上面的代码设置好了一切参数，但纹理还没有载入内存，下面的代码完成这个功能。

```

void Model::reloadTextures()
{
    for ( int i = 0; i < m_numMaterials; i++ )
        if ( strlen( m_pMaterials[i].m_pTextureFilename ) > 0 )
            m_pMaterials[i].m_texture = LoadGLTexture( m_pMaterials[i].m_pTextureFilename );
        else
            m_pMaterials[i].m_texture = 0;
}

```

有了数据，就可以写出绘制函数了，下面的函数根据模型的信息，按网格分组，分别绘制每一组的数据。

```
void Model::draw()
{
    GLboolean texEnabled = glIsEnabled( GL_TEXTURE_2D );

    // 按网格分组绘制
    for ( int i = 0; i < m_numMeshes; i++ )
    {

        int materialIndex = m_pMeshes[i].m_materialIndex;
        if ( materialIndex >= 0 )
        {
            glMaterialfv( GL_FRONT, GL_AMBIENT, m_pMaterials[materialIndex].m_ambient );
            glMaterialfv( GL_FRONT, GL_DIFFUSE, m_pMaterials[materialIndex].m_diffuse );
            glMaterialfv( GL_FRONT, GL_SPECULAR, m_pMaterials[materialIndex].m_specular );
            glMaterialfv( GL_FRONT, GL_EMISSION, m_pMaterials[materialIndex].m_emissive );
            glMaterialf( GL_FRONT, GL_SHININESS, m_pMaterials[materialIndex].m_shininess );

            if ( m_pMaterials[materialIndex].m_texture > 0 )
            {
                glBindTexture( GL_TEXTURE_2D, m_pMaterials[materialIndex].m_texture );
                glEnable( GL_TEXTURE_2D );
            }
            else
                glDisable( GL_TEXTURE_2D );
        }
        else
        {
            glDisable( GL_TEXTURE_2D );
        }

        glBegin( GL_TRIANGLES );
        {
            for ( int j = 0; j < m_pMeshes[i].m_numTriangles; j++ )
            {
                int triangleIndex = m_pMeshes[i].m_pTriangleIndices[j];
                const Triangle* pTri = &m_pTriangles[triangleIndex];

                for ( int k = 0; k < 3; k++ )
                {
                    int index = pTri->m_vertexIndices[k];

                    glNormal3fv( pTri->m_vertexNormals[k] );

```

```

        glTexCoord2f( pTri->m_s[k], pTri->m_t[k] );
        glVertex3fv( m_pVertices[index].m_location );
    }
}
glEnd();
}

if ( texEnabled )
    glEnable( GL_TEXTURE_2D );
else
    glDisable( GL_TEXTURE_2D );
}

```

有了上面的函数，我们来看看如何使用它们。首先，我们定义一个MilkshapeModel类。

Model \*pModel = NULL; // 定义一个指向模型类的指针

接着加载模型文件

```

pModel = new MilkshapeModel();
if ( pModel->loadModelData( "data/model.ms3d" ) == false )
{
    MessageBox( NULL, "不能加载data/model.ms3d文件", "加载错误", MB_OK | MB_ICONERROR );
    return 0;                                // 返回失败
}

```

接着载入纹理

```
pModel->reloadTextures();
```

完成了初始化操作，我们来实际绘制我们的模型

```
int DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 情况缓存
    glLoadIdentity();
    gluLookAt( 75, 75, 75, 0, 0, 0, 0, 1, 0 );

    glRotatef(yrot,0.0f,1.0f,0.0f);

    //绘制模型
    pModel->draw();

    yrot+=1.0f;
    return TRUE; //成功返回
}
```

简单吧？下一步我们该做什么？在以后的教程中，我将会加入骨骼动画的知识，到时候见吧！

#### 版权与使用声明：

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

#### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。



### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第30课

第32课 &gt;



## 第32课



拾取, Alpha混合, Alpha测试, 排序:

这又是一个小游戏, 交给的东西会很多, 慢慢体会吧

欢迎来到32课. 这课大概是在我所写作已来最大的一课. 超过1000 行代码和约1540行的HTML. 这也是第一课用到我新的NeHeGL 基本代码. 这课写了很长时间, 但我想他是值得期待的. 一些知识点用到是: Alpha 混合, Alpha 测试, 读取鼠标, 同时用到Ortho 和透视, 显示客户鼠标, 按深度排列物体, 动画帧从单张材质图 和更多要点, 你将学到更多精选的内容!

最初的版本是在屏幕上显示三个物体, 当你单击他们将改变颜色. 很有趣!?! 不怎样! 象往常一样, 我想给你们这些家伙留下一个超极好的课程. 我想使课程有趣, 丰富, 当然..美观. 所以, 经过几个星期的编码之后, 这课程完成了! 即使你不编码, 你仍会喜欢这课. 这是个完整的游戏. 游戏的目标是射击更多的靶子, 在你失去一定数的靶子后, 你将不能再用鼠标单击物体.

我确信会有批评, 但我非常乐观对这课! 我已在从深度里选择和排序物体这个主题里找到快乐!

一些需要注意的代码. 我仅仅会在lesson32.cpp里讨论. 有一些不成熟的改动在 NeHeGL 代码里. 最重要的改动是我加入鼠标支持在 WindowProc(). 我也加入 int mouse\_x, mouse\_y

在存鼠标运动. 在 NeHeGL.h 以下两条代码被加入: extern int mouse\_x; & extern int mouse\_y;

课程用到的材质是用 Adobe Photoshop 做的. 每个 .TGA 文件是一个 32 位图片有一个 alpha 通道. 若你不相信自己能在一个图片加入 alpha 通道, 找一本好书, 上网, 或读 Adobe Photoshop 帮助. 全部的过程非常相似, 我做了透明图在透明图课程. 调入你物体在 Adobe Photoshop (或一些其它图形处理程序, 且支持 alpha 通道). 用选择颜色工具选你图片的背景. 复制选区. 新建一个图. 粘贴生成新文件. 取消图片选择, 你图的背景应是黑色. 使周围是白色. 选全部图复制. 回到最初的图且建一个 alpha 通道. 粘贴黑和白透明图你就完成建立 alpha 通道. 存图片为 32 位 t.TGA 文件. 使确定保存透明背景是选中的, 保存!

如以往我希望你喜欢这课程. 我感兴趣你对他的想法. 若你有些问题或你发现一些问题, 告诉我. 我匆忙的完成这课程 所以若你发现哪部分很难懂, 给我发些邮件, 然后我会用不同的方式或更详细的解释!

```
#include <windows.h>
#include <stdio.h>
#include <stdarg.h>
#include <time.h>
#include "NeHeGL.h"
```

在第1课里, 我提倡关于适当的方法连接到 OpenGL 库. 在 Visual C++ 里点击 '项目', 设置, 连接项. 移下到 对象/库 模块 加入 OpenGL32.lib, GLu32.lib 和 GLaux.lib. 预编译一个需要的库的失败将使编译器找出所出的错误. 有时你不想发生! 使事情更坏, 若你仅仅预编译库在 debug 模式, 和有人试在 release 模式建立你程序... 更多的错误. 有许多人看代码. 他们大多数是新程序员. 他们取到你的代码, 试着编译. 他们得到错误, 删掉代码, 移走.

下面的代码告诉编译者去连接需要的库. 一点多些的字, 但少些以后的头痛. 在这个课程, 我们将连接 OpenGL32 库, GLu32 库 和 WinMM 库 (用来放音乐). 在这课程我们会调入 .TGA 文件, 所以我们不用 GLaux 库.

```
#pragma comment( lib, "opengl32.lib" )          // 在链接时连接 Opengl32.lib 库
#pragma comment( lib, "glu32.lib" )                // 链接 glu32.lib 库
#pragma comment( lib, "winmm.lib" )                // 链接 winmm.lib 库
```

下而的3行检查若 CDS\_FULLSCREEN 已被你的编译器定义. 若还没被定义, 我们给 CDS\_FULLSCREEN 为 4. 马上你完全部丢掉... 一些编译器不给 CDS\_FULLSCREEN 变量, 将返回一个错误, 但是 CDS\_FULLSCREEN 是有用的! 防止出错消息, 我们检查若 CDS\_FULLSCREEN 是否定义, 若出错, 我们定义他. 使每人生活更简单.

我们再定义 DrawTargets函数, 为窗口和按键设变量. 你若不懂定义, 读一遍MSDN术语表. 保持清醒, 我不是教 C/C++, 买一本好书若你对非gl代码要帮助!

```
#ifndef CDS_FULLSCREEN
#define CDS_FULLSCREEN 4
#endif
```

```
void DrawTargets();
```

```
GL_Window* g_window;
Keys* g_keys;
```

下面的代码是用户设置变量. base 是将用到的字体显示列表的开始列表值. roll 是将用到的移动的大地和建立旋转的云. level 应是级别 (我们开始是 1级). miss 保留失去了多少物体. 他还用来显示用户的士气(不丢失意味着高士气). kills 保留每级打到多少靶子. score 会保存运行时打中的总数, 同时用到结束比赛!

最后一行是让我们通过结构比较的函数. 是等待qsort 最后参数到 type (const \*void, const \*void).

// 用户定义的变量

```
GLuint base; // 字体显示列表
GLfloat roll; // 旋转的云
GLint level=1; // 现在的等级
GLint miss; // 丢失的数
GLint kills; // 打到的数
GLint score; // 当前的分数
bool game; // 游戏是否结束?
```

```
typedef int (*compfn)(const void*, const void*); // 定义用来比较的函数
```

现在为我们物体的结构. 这个结构存了所有一个物体的信息. 旋转的方向, 若被打中, 在屏幕的位置, 等等.

一个快速运动的变量... rot 我想让物体旋转特别的方向. hit 若物体没被打中将是 FALSE . 若物体给打中或飞出, 变量将是 TRUE.

变量frame 是用来存我们爆炸动画的周期. 每一帧改变增加一个爆炸材质. 在这课有更多在不久.

保存单个物体的移动方向, 我们用变量 dir. 一个dir 能有4 个值: 0 - 物体左移, 1 - 物体右移, 2 - 物体上移 和最后 3 - 物体下移

texid 能是从0到4的数. 0 表示是蓝面材质, 1 是水桶材质 , 2 是靶子的材质 , 3 是可乐的材质 和 4 是花瓶 材质. 最近在调入材质的代码, 你将看到先前5种材质来自目标图片.

x 和 y 两者都用来记屏模上物体的位置. x 表示物体在 x-轴, y 表示物体在 y-轴.

物体在z-轴上的旋转是记在变量spin. 在以后的代码, 我们将加或减spin基数在旅行的方向上.

最后, distance 保存我们物体到屏幕的距离. 距离是极端重要的变量, 我们将用他来计算屏幕的左右两边, 而且在对象关闭之前排序物体 , 画出物体的距离.

```
struct objects {  
    GLuint rot; // 旋转 (0-不转, 1-顺时针转, 2-逆时针转)  
    bool hit; // 物体碰撞?  
    GLuint frame; // 当前爆炸效果的动画帧  
    GLuint dir; // 物体的方向 (0-左, 1-右, 2-上, 3-下)  
    GLuint texid; // 物体材质 ID  
    GLfloat x; // 物体 X 位置  
    GLfloat y; // 物体 Y 位置  
    GLfloat spin; // 物体旋转  
    GLfloat distance; // 物体距离  
};
```

解释下面的代码没有真正的结果. 我们在这课调入TGA图代替bitmaps图片. 下面的用来表示TGA图片数据的结构是尽可能好的. 若你需要详细的解释下面的代码, 请读关于调入TGA文件的课程.

```
typedef struct // 新建一个结构
{
    GLubyte *imageData; // 图片数据(最大32位)
    GLuint bpp; // 图片颜色深度每象素
    GLuint width; // 图片宽度
    GLuint height; // 图片高度
    GLuint texID; // 贴图材质ID用来选择一个材质
} TextureImage; // 结构名称
```

紧接下面的代码为10个材质和30个物体留出空间. 若你打算在游戏里加更多物体, 得增加这个变量到你想到的数

```
TextureImage textures[10]; // 定义10个材质
objects object[30]; // 定义30个物体
```

我不想限制每个物体的大小. 我想瓶子(vase)比can高, 我想水桶bucket比瓶子宽. 去改变一切是简单的, 我建了一个结构存物体的宽和高.

我然后在最后一行代码中设每个物体的宽高. 得到这个coke cans的宽, 我将检查size[3].w. 蓝面是0, 水桶是1, 和靶子是2, 等. 宽度表现在w. 使有意义?

```
struct dimensions { // 物体维数
    GLfloat w; // 物体宽
    GLfloat h; // 物体高
};

// 每个物体的大小: 蓝面, 水桶, 靶子, 可乐, 瓶子
```

```
dimensions size[5] = { {1.0f,1.0f}, {1.0f,1.0f}, {1.0f,1.0f}, {0.5f,1.0f}, {0.75f,1.5f} };
```

下面是大段代码是调入我们 TGA 图片和转换他为材质. 这是同我在第25课所用的一样的代码 , 你可回去看一看.

我用TGA 图片的原因是他们是有alpha 通道的. 这个alpha 通道告诉 OpenGL 哪一部分图是透明的 , 哪一部分是白底. alpha 通道是被建立在图片处理程序, 并保存在.TGA图片里面. OpenGL 调入图片, 能用alpha 通道设置图片中每个象素透明的数量.

```
bool LoadTGA(TextureImage *texture, char *filename) // 调入一个TGA 文件到内存
{
    GLubyte    TGArheader[12]={0,0,2,0,0,0,0,0,0,0,0,0}; // (未)压缩的 TGA 头
    GLubyte    TGArcompare[12]; // 用来比较 TGA 头
    GLubyte    header[6]; // 首先 6 个有用的字节
    GLuint     bytesPerPixel; // 每象素字节数在 TGA 文件使用
    GLuint     imageSize; // 用来图片大小的存储
    GLuint     temp; // 临时变量
    GLuint     type=GL_RGBA; // 设置默认的 GL 模式为 RBGA

    FILE *file = fopen(filename, "rb"); // 打开 TGA 文件

    if( file==NULL || // 文件是否已存在 ?
        fread(TGArcompare,1,sizeof(TGArcompare),file)!=sizeof(TGArcompare) || // 是否读出12个字节?
        memcmp(TGArheader,TGArcompare,sizeof(TGArheader))!=0 || // 文件头是不是我们想要的?
        fread(header,1,sizeof(header),file)!=sizeof(header)) // 若正确则读下 6 个 Bytes
    {
        if (file == NULL) // 文件是否已存在 ?
            return FALSE; // 返回错误
        else // 否则
        {
            fclose(file); // 若有任何错误, 关掉文件
            return FALSE; // 返回错误
        }
    }

    texture->width = header[1] * 256 + header[0]; // 定义 TGA 宽
```

```

texture->height = header[3] * 256 + header[2];           // 定义 TGA 高

if(  texture->width <=0 ||                                // 若 宽<=0
    texture->height <=0 ||                                // 若 高<=0
    (header[4]!=24 && header[4]!=32))                  // 若 TGA 是 24 or 32 位?
{
    fclose(file);                                         // 若有任何错误, 关掉文件
    return FALSE;                                         // 返回错误
}

texture->bpp   = header[4];                                // 取 TGA 的位每象素 (24 或 32)
bytesPerPixel = texture->bpp/8;                            // 除以 8 得到字节每象素
imageSize       = texture->width*texture->height*bytesPerPixel; // 计算 所需内存为 TGA 数据

texture->imageData=(GLubyte *)malloc(imageSize);          // 分配 内存 为 TGA 数据

if(  texture->imageData==NULL ||                          // 这个内存是否存在?
    fread(texture->imageData, 1, imageSize, file)!=imageSize) // 图片大小与保留内存的大小
想等?
{
    if(texture->imageData!=NULL)                           // 图片数据的调入
        free(texture->imageData);                         // 若成功, 释放图象数据

    fclose(file);                                         // 关掉文件
    return FALSE;                                         // 返回错误
}

for(GLuint i=0; i<int(imageSize); i+=bytesPerPixel)        // 在图象数据里循环
{
    temp=texture->imageData[i];                          // 交换第1和第3 Bytes ( '红' red 和 '蓝' blue)
    texture->imageData[i] = texture->imageData[i + 2];    // 设 第1 Byte 得到变量 第3 Byte
    texture->imageData[i + 2] = temp;                     // 设第3 Byte 得到变量 'temp' (第1
Byte 变量)
}

fclose (file);                                         // 关掉文件

// 建立一个贴图材质从以上数据
glGenTextures(1, &texture[0].texID);                   // 生成 OpenGL 材质 ID

glBindTexture(GL_TEXTURE_2D, texture[0].texID);         // 绑定我们的材质
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); // 线过滤
器

```

```

glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR); // 线过
滤器

if (texture[0].bpp==24) // 若 TGA 是24 位的
{
    type=GL_RGB; // 设 ' type ' 为 GL_RGB
}

glTexImage2D(GL_TEXTURE_2D, 0, type, texture[0].width, texture[0].height, 0, type,
GL_UNSIGNED_BYTE, texture[0].imageData);

return true; // 材质建立成功, 返回正确
}

```

2D 字体材质代码同我已在前一课用的是一样的. 然而, 有一些小不同. 第一是你将看到仅仅唯一生成95 显示列表. 若你看字体材质, 你会看到只有 95 字母计算空间在图片顶, 左. 第二个事是你将通知分在16.0f 为 cx 和 我们只分在8.0f 为cy. 我这样做的结果是因为字体材质是256 象素宽, 但仅仅一伴就高(128 象素). 所以计算cx 我们分为16.0f 和计算分cy 为一半(8.0f).

若你不懂下面的代码, 回去读17课. 建立字体的代码的详细解释在第17课里!

```

GLvoid BuildFont(GLvoid) // 建立我们字体显示列表
{
    base=glGenLists(95); // 建立 95 显示列表
    glBindTexture(GL_TEXTURE_2D, textures[9].texID); // 绑我们字体材质
    for (int loop=0; loop<95; loop++) // 循环在 95 列表
    {
        float cx=float(loop%16)/16.0f; // X 位置 在当前字母
        float cy=float(loop/16)/8.0f; // Y 位置 在当前字母

        glNewList(base+loop,GL_COMPILE); // 开始建立一个列表
        glBegin(GL_QUADS); // 用四边形组成每个字母
        glTexCoord2f(cx, 1.0f-cy-0.120f); glVertex2i(0,0); // 质地 / 点 座标 (底 左)
        glTexCoord2f(cx+0.0625f, 1.0f-cy-0.120f); glVertex2i(16,0); // 质地 / 点 座标 (底 右)
        glTexCoord2f(cx+0.0625f, 1.0f-cy); glVertex2i(16,16); // 质地 / 点 座标 (顶 右)
        glTexCoord2f(cx, 1.0f-cy); glVertex2i(0,16); // 质地 / 点 座标 (顶 左)
    }
}

```

```

        glEnd();                                // 完成建立我们的四边形(字母)
        glTranslated(10,0,0);                   // 移到字体的右边
    glEndList();                            // 完成建军立这个显示列表
}                                         // 循环直到所有 256 完成建立
}

```

输出的代码也在第17课,但已修改为在屏幕输出我们的分数,等级和士气(不断改变的值).

```

GLvoid glPrint(GLint x, GLint y, const char *string, ...)           // 输出在屏幕的位置
{
    char      text[256];          // 保存在我们的字符串
    va_list   ap;                // 到列表的指针

    if (string == NULL)          // 若文字为空
        return;                  // 返回

    va_start(ap, string);       // 解析字符串
    vsprintf(text, string, ap); // 转换字符串
    va_end(ap);                // 结果的字符串

    glBindTexture(GL_TEXTURE_2D, textures[9].texID);           // 选择我们字体材质
    glPushMatrix();           // 存观看模式矩阵
    glLoadIdentity();         // 设观看模式矩阵
    glTranslated(x,y,0);      // 文字输出位置 (0,0 - 底 左-Bottom Left)
    glListBase(base-32);      // 选择字体设置
    glCallLists(strlen(text), GL_UNSIGNED_BYTE, text);        // 输出显示列表中的文字
    glPopMatrix();            // 取出以前的模式矩阵
}

```

这些代码调用排序程序. 它比较距离在两个结构并返回-1若第一个结构的距离小于第二个,1若 第一个结构的距离大于第二个0否则 (若 距离相等)

```

int Compare(struct objects *elem1, struct objects *elem2)           // 比较 函数
{
    if ( elem1->distance < elem2->distance)                      // 若 第一个结构的距离小于第二个

```

```

    return -1;                                // 返回 -1
else if (elem1->distance > elem2->distance)      // 若 第一个结构的距离大于第二个
    return 1;                                // 返回1
else                                         // 否则 (若 距离相等)
    return 0;                                // 返回 0
}

```

InitObject() 代码是来建立每个物体. 我们开始设 rot 为 1. 这使物体顺时针旋转. 然后设爆炸效果动画帧为0(我们不想爆炸效果从中间开始). 我们下面设 hit 为 FALSE, 意思是物体还没被击中或正开始. 选一个物体材质, texid 用来给一个随机的变量从 0 到 4. 0 是 blueface 材质 和4 是 vase 材质. 这给我们5种随机物体.

距离变量是在-0.0f to -40.0f (4000/100 is 40)的随机数 . 当我们真实的画物体, 我们透过在屏幕上的另10个单位. 所以当物体在画时, 他们将画从-10.0f to -50.0f 单位 在屏幕(不挨着, 也不离得太远). 我分随机数为 100.0f 得到更精确的浮点数值.

在给完随机的距离之后, 我们给物体一个随机的 y . 我们不想物体低于 -1.5f, 否则他将低于大地, 且我们不想物体高于3.0f. 所以留在我们的区间的随机数不能高于4.5f (-1.5f +4.5f=3.0f).

去计算 x 位置, 用一些狡猾的数学. 用我们的距离减去15.0f . 除以2 减5\*level. 再 减随机数 ( 0.0f 到5 ) 乘level. 减 5\*level random(0.0f to 5\*level) 这是最高级.

选一个方向.

使事情简单明白x, 写一个快的例子. 距离是 -30.0f , 当前级是 1:

```

object[num].x=(-30.0f-15.0f)/2.0f)-(5*1)-float(rand()%5);
object[num].x=(-45.0f/2.0f)-5-float(rand()%5);
object[num].x=(-22.5f)-5-{lets say 3.0f};
object[num].x=(-22.5f)-5-{3.0f};
object[num].x=-27.5f-{3.0f};
object[num].x=-30.5f;

```

开始在屏模上移 10 个单位 , 距离是 -30.0f. 其实是 -40.0f. 用透视的代码在 NeHeGL.cpp 文件.

GLvoid InitObject(int num) // 初始化一个物体

```
{
    object[num].rot=1;           // 顺时针旋转
    object[num].frame=0;         // 设爆炸效果动画帧为0
    object[num].hit=FALSE;       // 设点击检测为0
    object[num].texid=rand()%5;  // 设一个材质
    object[num].distance=-((float(rand()%4001)/100.0f)); // 随机距离
    object[num].y=-1.5f+(float(rand()%451)/100.0f);      // 随机 Y 位置
    // 随机开始 X 位置 基于物体的距离 和随机的延时量 (确定变量)
    object[num].x=((object[num].distance-15.0f)/2.0f)-(5*level)-float(rand()%(5*level));
    object[num].dir=(rand()%2);           // 选一个随机的方向
}
```

### 检查方向

```
if (object[num].dir==0) // 若随机的方向正确
{
    object[num].rot=2;           // 逆时针旋转
    object[num].x=-object[num].x; // 开始在左边 (否定 变量)
}
```

现在我们检查texid 来找出所选的的物体. 若 texid 为0, 所选的物体是 Blueface . blueface 总是在大地上面旋转. 确定开始时在地上的层, 我们设 y 是 -2.0f.

```
if (object[num].texid==0)           // 蓝色天空表面
    object[num].y=-2.0f;             // 总是在大地上面旋转
```

下面检查若texid 是 1. 这样, 电脑所选物体的是 Bucket. bucket不从左到右运动, 它从天上掉下来. 首先我们不得不设 dir 是 3. 这告诉电脑我们的水桶bucket 是掉下来或向下运动.

我们最初的代码假定物体从左到右运动. 因为bucket 是向下落的, 我们得不给它一个新的随机的变量 x . 若不是这样, bucket 会被看不到. 它将不在左边落下就在屏幕外面. 我们给它一个新的随机距离变量在屏幕上. 代替减去15, 我们仅仅减去 10. 这给我们一些幅度, 保持物体在屏幕?. 设我们的distance 是-30.0f, 从0.0f -40.0f的随机变量. 为什么从 0.0f 到 40.0f? 不是从0.0f to -40.0f? 答案很简单. rand() 函数总返回正数. 所以总是正数. 另外 , 回到我们的故事. 我们有个正数 从0.0f 到 40.0f. 我们加距离 最小 10.0f 除以 2. 举个例子 , 设x变量为 15 , 距离是 -30.0f:

```
object[num].x=float(rand()%int(-30.0f-10.0f))+((-30.0f-10.0f)/2.0f);
```

```
object[num].x=float(rand()%int(-40.0f)+(-40.0f)/2.0f);
```

```
object[num].x=float(15 {assuming 15 was returned})+(-20.0f);
```

```
object[num].x=15.0f-20.0f;
```

```
object[num].x=-5.0f;
```

下面设y. 我们想水桶从天上来. 我人不想穿过云. 所以我们设 y 为 4.5f. 刚在去的下面一点.

```
if (object[num].texid==1) // 水桶(Bucket)
{
    object[num].dir=3; // 下落
    object[num].x=float(rand()%int(object[num].distance-10.0f))+((object[num].distance-
10.0f)/2.0f);
    object[num].y=4.5f; // 随机 X, 开始在屏模上方
}
```

我们想靶子从地面突出到天上. 我们检查物体为 (texid 是 2). 若是, 设方向(dir) 是 2 (上). 用精确的数 x 位置.

我们不想target 开始在地上. 设 y 初值为 -3.0f (在地下). 然后减一个值从 0.0f 到 5 乘当前 level. 靶子不是立即出现. 在高级别是有延时, 通过 delay, 靶子将出现在一个在另一个以后, 给你很少时间打到他们.

```
if (object[num].texid==2) // 靶子
{
    object[num].dir=2; // 开始向上飞
    object[num].x=float(rand()%int(object[num].distance-10.0f))+((object[num].distance-10.0f)/2.0f);
    object[num].y=-3.0f-float(rand()%(5*level)); // 随机 X, 开始在下面的大地 + 随机变量
}
```

所有其它的物体从右到左旅行, 因而不必给任何变量付值来改变物体. 它们应该刚好工作在所给的随机变量.

现在来点有趣的材料! "为了alpha 混合技术正常的工作, 透明的原物必须不断地排定在从后向前画". 当画alpha 混合物体是, 在远处的物体是先画的, 这是非常重要的, 下面画紧临的上面的物体.

理由是简单的... Z 缓冲区防止 OpenGL 从已画好的混合东西再画象素. 这就是为什么会发生物体画在透明混合之后而不再显示出来. 为什么你最后看到的是一个四边形与物体重叠... 很不好看!

我们已知道每个物体的深度. 因而在初始化一个物体之后, 我们能通过把物体排序, 而用 qsort 函数(快速排序 sort), 来解决这个问题. 通过物体排序, 我们能确信第一个画的是最远的物体. 这意味着当我们画物体时, 起始于第一个物体, 物体通过用距离将被先画. 紧挨着那个物体(晚一会儿画) 将看到先前的物体在他们的后面, 再将适度的混合!

这文中的这行注释是我在 MSDN 里发现这些代码, 在网上花时间查找之后找到的解答. 他们工作的很好, 允许各种的排序结构. qsort 传送 4 个参数. 第一个参数指向物体数组(被排序的数组 d). 第二个参数是我们想排序数组的个数... 当然, 我们想所有的排序的物体普遍的被显示(各个 level). 第三个参数规定物体结构的大不, 第四个参数指向我们的 Compare() 函数.

大概有更好的排序结构的方法,但是 qsort() 工作起来... 快速方便,简单易用!

这个是重要的知识点,若你们想用 glAlphaFunc() 和 glEnable(GL\_ALPHA\_TEST), 排序是没必要的. 然而, 用Alpha 功能你被限制在完全透明或完全白底混合, 没有中间值. 用 Blendfunc()排序用一些更多的工作, 但他顾及半透明物体.

```
// 排序物体从距离:我们物体数组的开始地址 *** MSDN 代码修改为这个 TUT ***
// 各种的数据
// 各自的要素的
// 指针比较的函数
qsort((void *) &object, level, sizeof(struct objects), (compfn)Compare );
}
```

初始化的代码总是一样的. 首先的现两行取得我们window 的消息和我们建盘消息. 然后我们用 srand() 建一个基于时间的多样化的游戏. 之后我们调入 TGA 图片并用LoadTGA()转换到材质. 先前的 5个图片是将穿过屏幕的物体. Explode 是我们爆炸动画, 大地和天空 弥补现场背景, crosshair是你在屏幕上看到表现鼠标当前位置的十字光标, 最后, 用来显示分数, 标题和士气值的字体的图片. 若任何调入图片的失误, 则到返回 FALSE 值, 并程序结束. 值得注意的是这些基本代码不是返回整数型(INIT)的 FAILED 错误消息.

```
BOOL Initialize (GL_Window* window, Keys* keys) // 任何 OpenGL 从这初始化
{
    g_window = window;
    g_keys = keys;

    srand( (unsigned)time( NULL ) ); // 使随机化事件

    if ((!LoadTGA(&textures[0],"Data/BlueFace.tga")) || // 调入蓝面材质
        (!LoadTGA(&textures[1],"Data/Bucket.tga")) || // 调入水桶材质
        (!LoadTGA(&textures[2],"Data/Target.tga")) || // 调入靶子材质
        (!LoadTGA(&textures[3],"Data/Coke.tga")) || // 调入 可乐材质
        (!LoadTGA(&textures[4],"Data/Vase.tga")) || // 调入 花瓶 材质
        (!LoadTGA(&textures[5],"Data/Explode.tga")) || // 调入 爆炸材质
        (!LoadTGA(&textures[6],"Data/Ground.tga")) || // 调入 地面 材质
        (!LoadTGA(&textures[7],"Data/Sky.tga")) || // 调入 天空 材质
        (!LoadTGA(&textures[8],"Data/Crosshair.tga")) || // 调入 十字光标 材质
```

```

(!LoadTGA(&textures[9],"Data/Font.tga")))           // 调入 字符 材质
{
    return FALSE;                                // 若调入失败, 返回错误
}

```

若所有图片调入成功则轮到材质, 我们能继续初始化. 字体材质被调入, 因而保险能建立我们的字体. 我们跳入BuildFont()来做这些.

然后我们设置OpenGL. 背景色为黑, alpha 也设为0.0f. 深度缓冲区设为激活小于或等于测试.

glBlendFunc() 是很重要的一行代码. 我们设混合函数(GL\_SRC\_ALPHA, GL\_ONE\_MINUS\_SRC\_ALPHA). 这些加上alpha变量的屏幕上的混合物体存在物体的材质. 在设置混合模式之后, 我们激活blending(混合). 然后我们打开 2D 材质贴图, 最后, 打开 GL\_CULL\_FACE. 这是去除每个物体的后面( 没有一点浪费在一些我们看不到的循环 ). 画一些四边形逆时针卷动, 因而精致而适当的面片.

早先的教程我谈论使用glAlphaFunc()代替alpha 混合. 若你想用Alpha 函数, 注释出的两行混合代码和不注释的两行在glEnable(GL\_BLEND)之下. 你也能注释出qsort()函数在InitObject() 部分里的代码.

程序应该运行ok, 但sky 材质将不在这. 因为sky的材质已是一个alpha 变量0.5f. 当早在我谈关于Alpha函数, 我提及它只工作在alpha 变量0 或 1. 若你想它出现, 你将不得不修改sky的材质alpha 通道! 再则, 若你决定用Alpha 函数代替, 你不得排序物体. 两个方法都有好处! 再下而是从SGI 网站的快速引用:

"alpha 函数丢弃细节, 代替画他们在结构缓冲器里. 因此排序原来的物体不是必须的 (除了一些其它像混合alpha模式是打开的). 不占优势的是象素必须完全白底或完全透明".

BuildFont(); // 建立我们的字体显示列表

```

glClearColor(0.0f, 0.0f, 0.0f, 0.0f);           // 黑色背景
glClearDepth(1.0f);                            // 安装深度缓冲器
glDepthFunc(GL_LEQUAL);                        // 深度的类型测试
glEnable(GL_DEPTH_TEST);                       // 打开深度测试
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); // 打开 Alpha 混合

```

```

glEnable(GL_BLEND);           // 打开混合
glAlphaFunc(GL_GREATER,0.1f); // 设 Alpha 测试
glEnable(GL_ALPHA_TEST);     // 打开 Alpha 测试
glEnable(GL_TEXTURE_2D);      // 打开材质贴图
glEnable(GL_CULL_FACE);       // 去掉画物体的背面

```

在程序的这段, 还没有物体被定义, 所以循环30个物体 , 每个物体都调InitObject().

```

for (int loop=0; loop<30; loop++)           // 循环在 30 个物体Objects
    InitObject(loop);                      // 初始化每个物体

return TRUE;                                // 返回正确 (设初值成功)
}

```

在初始化代码里, 调入BuildFont() 建立95 的显示列表. 所以这里在程序结束前删掉95显示列表

```

void Deinitialize (void)           // 任何User 结束初始化从这
{
    glDeleteLists(base,95);        // 删掉所有95 字体显示列表
}

```

现在为急速原始物体... 是实际被选?形缓宓拇 ?. 第一行为我们选择物体的信息分配内存. hits 是当选择时碰撞迅检测的次数.

```

void Selection(void)             // 这是选择正确
{
    GLuint buffer[512];          // 设选择缓冲
    GLint hits;                  // 选择物体的数
}

```

下面的代码, 若游戏结束(FALSE). 没有选任何, 返回(exit). 若游戏还在运行 (TRUE), 用 Playsound() 命令放射击的声间. 仅仅调Selection()的时间是在当已鼠标按下时和每次按下按键调用时, 想放射击的声音. 声音在放在 async 模式 ,所以放音乐是程序不会停.

```
if (game) // 游戏是否结束?  
    return; // 是 , 返回, 不在检测 Hits  
  
PlaySound("data/shot.wav",NULL,SND_ASYNC); // 放音乐 Gun Shot
```

设视点. `viewport[]` 包括当前 `x, y`, 当前的视点(OpenGL Window)长度 , 宽度.

`glGetIntegerv(GL_VIEWPORT, viewport)` 取当前视点存在`viewport[]`. 最初 , 等于 OpenGL 窗口维数. `glSelectBuffer(512, buffer)` 说 OpenGL 用这个内存.

```
// 视点的大小. [0] 是 <x>, [1] 是 <y>, [2] 是 <length>, [3] 是 <width>  
GLint viewport[4];  
  
// 这是设视点的数组在屏幕窗口的位置  
glGetIntegerv(GL_VIEWPORT, viewport);  
glSelectBuffer(512, buffer); // 告诉 OpenGL 使我们的数组来选择
```

存openGl的模式. 在这个模式什么也不画. 代替, 在选择模式物体渲染信息存在缓存.

下面初实化name 堆栈,通过调入`glInitNames()` 和`glPushName(0)`. |它重要的是标记若程序不在选择模式, 一个到`glPushName()`调用将忽略. 当然在选择的模试, 但这一些是是紧记的.

```
// 设 OpenGL 选择模式. 将不画东西. 物体 ID ' 的广度放在内存  
(void) glRenderMode(GL_SELECT);  
  
glInitNames(); // 设名字堆栈
```

```
glPushName(0); // Push 0 (最少一个) 在栈上
```

之后, 不得不限制在光标的下面画图. 为了做这些得用到投影矩阵. 然后把它推到堆栈中. 重设矩阵则用到 glLoadIdentity().

用 gluPickMatrix() 限制的画. 第1个参数是当前鼠标的 x-座标, 第2个参数是当前鼠标的 y-座标, 然后宽和高的选区. 最后当前的 viewport[]. viewport[] 是指出视点的边界. x 和 \_y 将在选区的中心.

```
glMatrixMode(GL_PROJECTION); // 选投影矩阵
glPushMatrix(); // 压入投影矩阵
glLoadIdentity(); // 重设矩阵

// 这是建一个矩阵使鼠标在屏幕缩放
gluPickMatrix((GLdouble) mouse_x, (GLdouble) (viewport[3]-mouse_y), 1.0f, 1.0f, viewport);
```

调入 gluPerspective() 应用透视矩阵, 被 gluPickMatrix() 选择矩阵限制所画区域.

打开 modelview 矩阵, 调用 DrawTargets() 画我们的靶子. 画靶子在 DrawTargets() 而不在 Draw() 是因为仅仅想选择物体的碰撞检测且, 不是天空, 大地, 光标.

之后, 打开回到发射矩阵, 从堆栈中弹出矩阵. 之扣打开回到 modelview 矩阵.

最后一行, 回到渲染模式 因而物体画的很真实的在屏幕上. hits 将采集 gluPickMatrix() 所需要取渲染的物体数.

```
// 应用透视矩阵
gluPerspective(45.0f, (GLfloat) (viewport[2]-viewport[0])/(GLfloat) (viewport[3]-viewport[1]), 0.1f,
100.0f);
glMatrixMode(GL_MODELVIEW); // 选择模型变换矩阵
DrawTargets(); // 画目标
glMatrixMode(GL_PROJECTION); // 选择投影变换矩阵
glPopMatrix(); // 取出投影矩阵
glMatrixMode(GL_MODELVIEW); // 选模式显示矩阵
```

```
hits=glRenderMode(GL_RENDER);
```

// 切换模式, 找出有多少

检查若多于0个hits 记录. 若这样, 设choose 为 第一个物体的名子. depth 取得它有多远.

每个hit 分有4个项目在内存. 第一, 在名子堆栈上打击发生时的数字. 第二, 所选物体的最小z值. 第三, 所选物体的最大 z 值, 最后, 在同一时间里所选物体名子堆栈的内容(物体的名子). 在这一课, 我们仅对最小z值和物体名子感兴趣.

```
if (hits > 0) // 若有大于0个 Hits
{
    int choose = buffer[3]; // 选择第一物体
    int depth = buffer[1]; // 存它有多远
```

做循环所有hits 使没有物体在第一个物体旁边. 否则, 两个物体会重叠, 一个物体碰到另一个. 当你射击时, 重叠的物体会被误选.

```
for (int loop = 1; loop < hits; loop++)
{
    // 对于其它的物体
    if (buffer[loop*4+1] < GLuint(depth))
    {
        choose = buffer[loop*4+3]; // 选择更近的物体
        depth = buffer[loop*4+1]; // 保存它有多远
    }
}
```

若物体被选.

```
if (!object[choose].hit) // 如果物体还没有被击中
```

```
{  
    object[choose].hit=TRUE;           // 标记物体象被击中  
    score+=1;                         // 增加分数  
    kills+=1;                          // 加被杀数
```

如下

```
if (kills>level*5)          // 已有新的级?  
{  
    miss=0;                   // 失掉数回0  
    kills=0;                  // 设 Kills 数为 0  
    level+=1;                 // 加 Level  
    if (level>30)             // 高过 30?  
        level=30;              // 设 Level 为 30 (你是 God 吗?)  
    }  
}  
}  
}
```

如下

```
void Update(DWORD milliseconds)           // 这里用来更新  
{  
    if (g_keys->keyDown[VK_ESCAPE])        // 按下 ESC?  
    {  
        TerminateApplication (g_window);   // 推出程序  
    }
```

如下

```

if (g_keys->keyDown[' '] && game)           // 按下空格键?
{
    for (int loop=0; loop<30; loop++)          // 循环所有的物体
        InitObject(loop);                      // 初始化

    game=FALSE;                                // 设game为false
    score=0;                                    // 分数为0
    level=1;                                    // 级别为1
    kills=0;                                    // 杀敌数为0
    miss=0;                                    // 漏过数为0
}

if (g_keys->keyDown[VK_F1])                  // 按下f1?
{
    ToggleFullscreen (g_window);              // 换到全屏模式
}

roll-=milliseconds*0.00005f;                  // 云的旋转

for (int loop=0; loop<level; loop++)          // 循环所有的物体
{

```

下面的代码按物体的运动方向更新所有的运动

```

if (object[loop].rot==1)
    object[loop].spin-=0.2f*(float(loop+milliseconds)); // 若顺时针,则顺时针旋转

if (object[loop].rot==2)
    object[loop].spin+=0.2f*(float(loop+milliseconds)); // 若逆时针,则逆时针旋转

if (object[loop].dir==1)
    object[loop].x+=0.012f*float(milliseconds);        // 向右移动

if (object[loop].dir==0)
    object[loop].x-=0.012f*float(milliseconds);        // 向左移动

if (object[loop].dir==2)

```

```
object[loop].y+=0.012f*float(milliseconds);      // 向上移动

if (object[loop].dir==3)
    object[loop].y-=0.0025f*float(milliseconds);      // 向下移动
```

下面的代码处理当物体移动到边缘处,如果你没有击中它的结果

```
// 如果到达左边界,你没有击中,则增加丢失的目标数
if ((object[loop].x<(object[loop].distance-15.0f)/2.0f) && (object[loop].dir==0) && !object[loop].hit)
{
    miss+=1;
    object[loop].hit=TRUE;
}

// 如果到达右边界,你没有击中,则增加丢失的目标数
if ((object[loop].x>-(object[loop].distance-15.0f)/2.0f) && (object[loop].dir==1) && !object[loop].hit)
{
    miss+=1;
    object[loop].hit=TRUE;
}

// 如果到达下边界,你没有击中,则增加丢失的目标数
if ((object[loop].y<-2.0f) && (object[loop].dir==3) && !object[loop].hit)
{
    miss+=1;
    object[loop].hit=TRUE;
}

//如果到达左边界,你没有击中,则方向变为向下
if ((object[loop].y>4.5f) && (object[loop].dir==2))
    object[loop].dir=3;
```

下面的代码在屏幕上绘制一个图像

```
void Object(float width,float height,GLuint texid)           // 画物体用需要的宽 , 高 , 材质
{
    glBindTexture(GL_TEXTURE_2D, textures[texid].texID);      // 选合适的材质
    glBegin(GL_QUADS);                                         // 开始画四边形
        glTexCoord2f(0.0f,0.0f); glVertex3f(-width,-height,0.0f);
        glTexCoord2f(1.0f,0.0f); glVertex3f( width,-height,0.0f);
        glTexCoord2f(1.0f,1.0f); glVertex3f( width, height,0.0f);
        glTexCoord2f(0.0f,1.0f); glVertex3f(-width, height,0.0f);
    glEnd();
}
```

下面的代码绘制爆炸的效果

```
void Explosion(int num)           // 画爆炸动画的1帧
{
    float ex = (float)((object[num].frame/4)%4)/4.0f;          // 计算爆炸时生成的x的纹理坐标
    float ey = (float)((object[num].frame/4)/4)/4.0f;          // 计算爆炸时生成的y的纹理坐标

    glBindTexture(GL_TEXTURE_2D, textures[5].texID);           // 选择爆炸的纹理
    glBegin(GL_QUADS);
        glTexCoord2f(ex ,1.0f-(ey )); glVertex3f(-1.0f,-1.0f,0.0f);
        glTexCoord2f(ex+0.25f,1.0f-(ey )); glVertex3f( 1.0f,-1.0f,0.0f);
        glTexCoord2f(ex+0.25f,1.0f-(ey+0.25f)); glVertex3f( 1.0f, 1.0f,0.0f);
        glTexCoord2f(ex ,1.0f-(ey+0.25f)); glVertex3f(-1.0f, 1.0f,0.0f);
    glEnd();
```

增加帧数,如果大于63,则重置动画

```

object[num].frame+=1;           // 加当前的爆炸动画帧
if (object[num].frame>63)       // 是否已完成所有的16帧?
{
    InitObject(num);          // 定义物体(给新的变量)
}
}

```

## 画靶子

```

void DrawTargets(void)           // 画靶子
{
    glLoadIdentity();           // 移入屏幕 20 个单位
    glTranslatef(0.0f,0.0f,-10.0f); // 循环在 9 个物体
    for (int loop=0; loop<level; loop++)
    {
        glLoadName(loop);      // 给物体新名字
        glPushMatrix();         // 存矩阵
        glTranslatef(object[loop].x,object[loop].y,object[loop].distance); // 物体的位置 (x,y)

        if (object[loop].hit)    // 若物体已被点击
        {
            Explosion(loop);   // 画爆炸动画
        }
        else
        {
            glRotatef(object[loop].spin,0.0f,0.0f,1.0f); // 旋转物体
            Object(size[object[loop].texid].w,size[object[loop].texid].h,object[loop].texid); // 画物体
        }
        glPopMatrix();           // 弹出矩阵
    }
}

```

下面的代码绘制整个场景

```
void Draw(void) // 画我们的现场
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除屏幕和深度缓
冲
    glLoadIdentity(); // 重设矩阵
```

下面的代码绘制飘动的天空,它由四块纹理组成,每一块的移动速度都不一样,并把它们混合起来

```
glPushMatrix();
glBindTexture(GL_TEXTURE_2D, textures[7].texID); // 选天空的材质
glBegin(GL_QUADS);
    glTexCoord2f(1.0f,roll/1.5f+1.0f); glVertex3f( 28.0f,+7.0f,-50.0f);
    glTexCoord2f(0.0f,roll/1.5f+1.0f); glVertex3f(-28.0f,+7.0f,-50.0f);
    glTexCoord2f(0.0f,roll/1.5f+0.0f); glVertex3f(-28.0f,-3.0f,-50.0f);
    glTexCoord2f(1.0f,roll/1.5f+0.0f); glVertex3f( 28.0f,-3.0f,-50.0f);

    glTexCoord2f(1.5f,roll+1.0f); glVertex3f( 28.0f,+7.0f,-50.0f);
    glTexCoord2f(0.5f,roll+1.0f); glVertex3f(-28.0f,+7.0f,-50.0f);
    glTexCoord2f(0.5f,roll+0.0f); glVertex3f(-28.0f,-3.0f,-50.0f);
    glTexCoord2f(1.5f,roll+0.0f); glVertex3f( 28.0f,-3.0f,-50.0f);

    glTexCoord2f(1.0f,roll/1.5f+1.0f); glVertex3f( 28.0f,+7.0f,0.0f);
    glTexCoord2f(0.0f,roll/1.5f+1.0f); glVertex3f(-28.0f,+7.0f,0.0f);
    glTexCoord2f(0.0f,roll/1.5f+0.0f); glVertex3f(-28.0f,+7.0f,-50.0f);
    glTexCoord2f(1.0f,roll/1.5f+0.0f); glVertex3f( 28.0f,+7.0f,-50.0f);

    glTexCoord2f(1.5f,roll+1.0f); glVertex3f( 28.0f,+7.0f,0.0f);
    glTexCoord2f(0.5f,roll+1.0f); glVertex3f(-28.0f,+7.0f,0.0f);
    glTexCoord2f(0.5f,roll+0.0f); glVertex3f(-28.0f,+7.0f,-50.0f);
    glTexCoord2f(1.5f,roll+0.0f); glVertex3f( 28.0f,+7.0f,-50.0f);
glEnd();
```

下面的代码绘制地面

```
glBindTexture(GL_TEXTURE_2D, textures[6].texID);           // 大地材质  
glBegin(GL_QUADS);  
    glTexCoord2f(7.0f,4.0f-roll); glVertex3f( 27.0f,-3.0f,-50.0f);  
    glTexCoord2f(0.0f,4.0f-roll); glVertex3f(-27.0f,-3.0f,-50.0f);  
    glTexCoord2f(0.0f,0.0f-roll); glVertex3f(-27.0f,-3.0f,0.0f);  
    glTexCoord2f(7.0f,0.0f-roll); glVertex3f( 27.0f,-3.0f,0.0f);  
glEnd();
```

### 绘制我们的靶子

```
DrawTargets();           // 画我们的靶子  
glPopMatrix();
```

### 下面的代码绘制我们的十字光标

```
// 十字光标 (在光标里)  
RECT window;           // 用来存窗口位置  
GetClientRect (g_window->hWnd,&window);           // 取窗口位置  
glMatrixMode(GL_PROJECTION);  
glPushMatrix();  
glLoadIdentity();  
glOrtho(0>window.right,0>window.bottom,-1,1);           // 设置为正投影  
glMatrixMode(GL_MODELVIEW);  
glTranslated(mouse_x>window.bottom-mouse_y,0.0f);           // 移动到当前鼠标位置  
Object(16,16,8);           // 画十字光标
```

### 下面的代码用来显示帮助文字

```
// 游戏状态 / 标题名称
glPrint(240,450,"NeHe Productions");           // 输出 标题名称
glPrint(10,10,"Level: %i",level);               // 输出 等级
glPrint(250,10,"Score: %i",score);              // 输出 分数
```

如果丢失10个物体,游戏结束

```
if (miss>9)                                // 我们已丢失 10 个物体?
{
    miss=9;                                  // 限制丢失是10个
    game=TRUE;                               // 游戏结束
}
```

在下面的代码里, 我们查看若game 是TRUE. 若 game 是TRUE, 我们输出 ' GAME OVER ' 游戏结束的消息. 若game 是false, 我们输出 玩家的士气morale (到10溢出). 士气 morale是被设计用来从10减去玩家失误的次数(miss) . 玩家失掉的越多, 士气越低.

```
if (game)                                    // 游戏是否结束?
    glPrint(490,10,"GAME OVER");             // 结束消息
else
    glPrint(490,10,"Morale: %i/10",10-miss); // 输出剩余生命
```

最后做的事我们选投影矩阵, 恢复(取出)我们的矩阵返回到前一个情形, 设矩阵模式为 modelview , 刷新缓冲区 , 使所有物体被渲染.

```
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glMatrixMode(GL_MODELVIEW);
```

```
glFlush();  
}
```

这课程是多次熬夜的成果,许多的时间用来编码和写 HTML. 在这一课结束的时候你应你会学会怎样picking, sorting, alpha blending and alpha testing 工作. 制做点和软件. 每一个游戏, 到精选的GUI '们. 最好的未来是制做时你不用记录物体. 你给一个名字和碰撞. 这很简单! 用alpha 通道和alpha 测试你能使物体完全显示, 或漏出一些. 结果是很好, 你不用担心关于显示物体的材质, 除非你不显示他们! 同以往一样, 我希望你喜欢这个课程, 愿看到一些好的游戏或好的项目从这个课程诞生. 如果你有什么问题或找到错误, 让我知道 ... 我仅是一个普通人 :)

我将花大量的时间加入东西像物理系统, 更多图, 更多声音, 等. 虽然只是一个课程! 我不写不按车灯和车轮. 我写这个用尽量不混乱的方法教你 OpenGL. 我希望看到一些严谨的修改. 若你找一些cool的课程发给我一份. 若是好的修改我将放到下载页. 若有足够充分的修改我会专注修改这个课程的版本! 我在这里给你一个起点. 剩下的靠你了 :)

要点: 这是很重要的, 称为glTexImage2D 你设为两种格式国际 GL\_RGBA. 否则 alpha blending 将不工作!

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩, 在网络上我以DancingWind为昵称, 我的联系方式是zhouwei02@mails.tsinghua.edu.cn, 如果你有任何问题, 都可以联系我。

#### 引子

网络是一个共享的资源, 但我在自己的学习生涯中浪费大量的时间去搜索可用的资料, 在现实生活中花费了大量的金钱和时间在书店中寻找资料, 于是我给自己起了个昵称DancingWind, 其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后, 我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容, 大多都来自共享的资源, 所以我没有资格把它们据为己有, 或声称自己为这些资源作出了一点贡献。故任何人都可以复制, 修改, 重新发表, 甚至以自己的名义发表, 我



都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第31课

第33课 >



## 第33课



加载压缩和未压缩的TGA文件:

在这一课里，你将学会如何加载压缩和为压缩的TGA文件，由于它使用RLE压缩，所以非常的简单，你能很快地熟悉它的。

我见过很多人在游戏开发论坛或其它地方询问关于TGA读取的问题。接下来的程序及注释将会向你展示如何读取未压缩的TGA文件和RLE压缩的文件。这个详细的教程适合于OpenGL，但是我计划改进它使其在将来更具普遍性。

我们将从两个头文件开始。第一个文件控制纹理结构，在第二个里，结构和变量将为程序读取所用。

就像每个头文件那样，我们需要一些包含保护措施以防止文件被重复包含。

在文件的顶部加入这样几行程序：

```
#ifndef __TEXTURE_H__           // 看看此头文件是否已经被包含  
#define __TEXTURE_H__          // 如果没有，定义它
```

然后滚动到程序底部并添加：

```
#endif // __TEXTURE_H__ 结束包含保护
```

这三行程序防止此文件被重复包含。文件中剩下的代码将处于这头两行和这最后一行之间。

在这个头文件中，我们将要加入完成每件工作所需的标准头文件。在#define \_\_TGA\_H\_\_后添加如下几行：

```
#pragma comment(lib, "OpenGL32.lib") // 链接 OpenGL32.lib
#include <windows.h> // 标准 Windows 头文件
#include <stdio.h> // 标准文件 I/O 头文件
#include <gl\gl.h> // 标准 OpenGL 头文件
```

第一个头文件是标准Windows头文件，第二个是我们稍后的文件I/O所准备的，第三个是OpenGL32.lib所需的标准OpenGL头文件。

我们将需要一块空间存储图像数据以及OpenGL生成纹理所需的类型。我们将要用到以下结构：

```
typedef struct
{
    GLubyte* imageData; // 控制整个图像的颜色值
    GLuint bpp; // 控制单位像素的 bit 数
    GLuint width; // 整个图像的宽度
    GLuint height; // 整个图像的高度
    GLuint texID; // 使用 glBindTexture 所需的纹理 ID.
    GLuint type; // 描述存储在 *ImageData 中的数据 (GL_RGB 或 GL_RGBA)
} Texture;
```

现在说说其它的，更长的头文件。同样我们需要一些包含保护措施，这和上述最后一个是一样的。

接下来，看看另外两个结构，它们将在处理TGA文件的过程中使用。

```
typedef struct
{
    GLubyte Header[12];           // 文件头决定文件类型
} TGAHeader;

typedef struct
{
    GLubyte header[6];           // 控制前6个字节
    GLuint bytesPerPixel;        // 每像素的字节数 (3 或 4)
    GLuint imageSize;             // 控制存储图像所需的内存空间
    GLuint type;                 // 图像类型 GL_RGB 或 GL_RGBA
    GLuint Height;                // 图像的高度
    GLuint Width;                 // 图像宽度
    GLuint Bpp;                   // 每像素的比特数 (24 或 32)
} TGA;
```

现在我们声明那两个结构的一些实例，那样我们可以在程序中使用它们。

```
TGAHeader tgaheader;          // 用来存储我们的文件头
TGA tga;                      // 用来存储文件信息
```

我们需要定义一对文件头，那样我们能够告诉程序什么类型的文件头处于有效的图像上。如果是未压缩的TGA图像，前12字节将会是002000000000，如果是RLE压缩的，则是001000000000。这两个值允许我们检查正在读取的文件是否有效。

```
// 未压缩的TGA头
GLubyte uTGACcompare[12] = {0,0,2,0,0,0,0,0,0,0,0,0};
```

```
// 压缩的TGA头  
GLubyte cTGACompare[12] = {0,0,10,0,0,0,0,0,0,0,0,0};
```

最后，我们声明两个函数用于读取过程。

```
// 读取一个未压缩的文件  
bool LoadUncompressedTGA(Texture *, char *, FILE *);  
// 读取一个压缩的文件  
bool LoadCompressedTGA(Texture *, char *, FILE *);
```

现在，回到cpp文件，和程序中真正首当其冲部分，我将会省去一些错误消息处理代码并且使教程更短、更具可读性。你可以参看教程包含的文件（在文章的尾部有链接）。

马上，我们就可以在文件开头包含我们刚刚建立的头文件。

```
#include "tga.h" // 包含我们刚刚建立的头文件
```

不我们不需要包含其它任何文件了，因为我们已经在自己刚刚完成的头文件中包含他们了。

接下来，我们要做的事情是看看第一个函数，名为LoadTGA(...).

```
// 读取一个TGA文件!  
bool LoadTGA(Texture * texture, char * filename)  
{
```

它有两个参数。前者是一个指向纹理结构的指针，你必须在你的代码中声明它（见包含的例子）。后者是一个字符串，它告诉计算机在哪里去找你的纹理文件。

函数的前两行声明了一个文件指针，然后打开由“filename”参数指定的文件，它由函数的第二个指针传递进去。

```
FILE * fTGA;           // 声明文件指针
fTGA = fopen(filename, "rb"); // 以读模式打开文件
```

接下来的几行检查指定的文件是否已经正确地打开。

```
if(fTGA == NULL)           // 如果此处有错误
{
    ...Error code...
    return false;           // 返回 False
}
```

下一步，我们尝试读取文件的首12个字节的内容并且将它们存储在我们的TGAHeader结构中，这样，我们得以检查文件类型。如果fread失败，则关闭文件，显示一个错误，并且函数返回false。

```
if(fread(&tgaheader, sizeof(TGAHeader), 1, fTGA) == 0)
{
    ...Error code here...
    return false;           // 如果失败则返回 False
}
```

接着，通过我们用辛苦编的程序刚读取的头，我们继续尝试确定文件类型。这可以告诉我们它是压缩的、未压缩甚至是错误的文件类型。为了达到这个目的，我们将会使用memcmp(...)函数。

```
// 如果文件头附合未压缩的文件头格式
if(memcmp(uTGAcompare, &tgaheader, sizeof(tgaheader)) == 0)
{
    // 读取未压缩的TGA文件
    LoadUncompressedTGA(texture, filename, fTGA);
}

// 如果文件头附合压缩的文件头格式
else if(memcmp(cTGAcompare, &tgaheader, sizeof(tgaheader)) == 0)
{
    // 读取压缩的TGA格式
    LoadCompressedTGA(texture, filename, fTGA);
}

else // 如果任一个都不符合
{
    ...Error code here...
    return false; // 返回 False
}
```

我们将要开始读取一个未压缩格式文件的章节。

下面开始我们要做的第一件事，像往常一样，是函数头。

```
//读取未压缩的TGA文件
bool LoadUncompressedTGA(Texture * texture, char * filename, FILE * fTGA)
{
```

这个函数有3个参数。头两个和LoadTGA中的一样，仅仅是简单的传递。第三个是来自前一个函数中的文件指针，因此我们没有丢失我们的空间。

接下来我们试着再从文件中读取6个字节的内容，并且存储在tga.header中。如果他失败了，我们运行一些错误处理代码，并且返回false。

```
// 尝试继续读取6个字节的内容
if(fread(tga.header, sizeof(tga.header), 1, fTGA) == 0)
{
```

```

...Error code here...
return false;           // 返回 False
}

```

现在我们有了计算图像的高度、宽度和BPP的全部信息。我们在纹理和本地结构中都将存储它。

```

texture->width = tga.header[1] * 256 + tga.header[0]; // 计算高度
texture->height = tga.header[3] * 256 + tga.header[2]; // 计算宽度
texture->bpp = tga.header[4];           // 计算BPP
tga.Width = texture->width;           // 拷贝Width到本地结构中去
tga.Height = texture->height;          // 拷贝Height到本地结构中去
tga.Bpp = texture->bpp;               // 拷贝Bpp到本地结构中去

```

现在，我们需要确认高度和宽度至少为1个像素，并且bpp是24或32。如果这些值中的任何一个超出了它们的界限，我们将再一次显示一个错误，关闭文件，并且离开此函数。

```

// 确认所有的信息都是有效的
if((texture->width <= 0) || (texture->height <= 0) || ((texture->bpp != 24) && (texture->bpp != 32)))
{
    ...Error code here...
    return false;           // 返回 False
}

```

接下来我们设置图像的类型。24 bit图像是GL\_RGB，32 bit 图像是GL\_RGBA

```

if(texture->bpp == 24)           // 是24 bit图像吗 ?
{
    texture->type = GL_RGB;      //如果是，设置类型为GL_RGB
}
else                            // 如果不是24bit,则必是32bit
{

```

```
texture->type = GL_RGBA; //这样设置类型为GL_RGBA  
}
```

现在我们计算每像素的字节数和总共的图像数据。

```
tga.bytesPerPixel = (tga.Bpp / 8); // 计算BPP  
// 计算存储图像所需的内存  
tga.imageSize = (tga.bytesPerPixel * tga.Width * tga.Height);
```

我们需要一些空间去存储整个图像数据，因此我们将要使用malloc分配正确的内存数量

然后我们确认内存已经分配，并且它不是NULL。如果出现了错误，则运行错误处理代码。

```
// 分配内存  
texture->imageData = (GLubyte *)malloc(tga.imageSize);  
if(texture->imageData == NULL) // 确认已经分配成功  
{  
    ...Error code here...  
    return false; // 确认已经分配成功  
}
```

这里我们尝试读取所有的图像数据。如果不能，我们将再次触发错误处理代码。

```
// 尝试读取所有图像数据  
if(fread(texture->imageData, 1, tga.imageSize, fTGA) != tga.imageSize)  
{  
    ...Error code here...  
    return false; // 如果不能，返回false  
}
```

TGA文件用逆OpenGL需求顺序的方式存储图像，因此我们必须将格式从BGR到RGB。为了达到这一点，我们交换每个像素的第一个和第三个字节的内容。

Steve Thomas补充：我已经编写了能稍微更快速读取TGA文件的代码。它涉及到仅用3个二进制操作将BGR转换到RGB的方法。

然后我们关闭文件，并且成功退出函数。

```
// 开始循环
for(GLuint cswap = 0; cswap < (int)tga.imageSize; cswap += tga.bytesPerPixel)
{
    // 第一字节 XOR 第三字节 XOR 第一字节 XOR 第三字节
    texture->imageData[cswap] ^= texture->imageData[cswap+2] ^=
        texture->imageData[cswap] ^= texture->imageData[cswap+2];
}

fclose(fTGA);           // 关闭文件
return true;            // 返回成功
}
```

以上是读取未压缩型TGA文件的方法。读取RLE压缩型文件的步骤稍微难一点。我们像平时一样读取文件头并且收集高度 / 宽度 / 色彩深度，这和读取未压缩版本是一致的。

```
bool LoadCompressedTGA(Texture * texture, char * filename, FILE * fTGA)
{
    if(fread(tga.header, sizeof(tga.header), 1, fTGA) == 0)
    {
        ...Error code here...
    }
    texture->width = tga.header[1] * 256 + tga.header[0];
    texture->height = tga.header[3] * 256 + tga.header[2];
    texture->bpp = tga.header[4];
    tga.Width = texture->width;
    tga.Height = texture->height;
    tga.Bpp = texture->bpp;
    if((texture->width <= 0) || (texture->height <= 0) || ((texture->bpp != 24) && (texture->bpp !
```

```
=32))  
{  
    ...Error code here...  
}  
tga.bytesPerPixel = (tga.Bpp / 8);  
tga.imageSize = (tga.bytesPerPixel * tga.Width * tga.Height);
```

现在我们需要分配存储图像所需的空间，这是为我们解压缩之后准备的，我们将使用 malloc。如果内存分配失败，运行错误处理代码，并且返回false。

```
// 分配存储图像所需的内存空间  
texture->imageData = (GLubyte *)malloc(tga.imageSize);  
if(texture->imageData == NULL) // 如果不能分配内存  
{  
    ...Error code here...  
    return false; // 返回 False  
}
```

下一步我们需要决定组成图像的像素数。我们将它存储在变量“pixelcount”中。

我们也需要存储当前所处的像素，以及我们正在写入的图像数据的字节，这样避免溢出写入过多的旧数据。

我们将要分配足够的内存来存储一个像素。

```
GLuint pixelcount = tga.Height * tga.Width; // 图像中的像素数  
GLuint currentpixel = 0; // 当前正在读取的像素  
GLuint currentbyte = 0; // 当前正在向图像中写入的像素  
// 一个像素的存储空间  
GLubyte * colorbuffer = (GLubyte *)malloc(tga.bytesPerPixel);
```

接下来我们将要进行一个大循环。

让我们将它分解为更多可管理的块。

首先我们声明一个变量来存储“块”头。块头指示接下来的段是RLE还是RAW，它的长度是多少。如果一字节头小于等于127，则它是一个RAW头。头的值是颜色数，是负数，在我们处理其它头字节之前，我们先读取它并且拷贝到内存中。这样我们将我们得到的值加1，然后读取大量像素并且将它们拷贝到ImageData中，就像我们处理未压缩型图像一样。如果头大于127，那么它是下一个像素值随后将要重复的次数。要获取实际重复的数量，我们将它减去127以除去1bit的头标示符。然后我们读取下一个像素并且依照上述次数连续拷贝它到内存中。

```
do          // 开始循环
{
    GLubyte chunkheader = 0;           // 存储Id块值的变量
    if(fread(&chunkheader, sizeof(GLubyte), 1, fTGA) == 0) // 尝试读取块的头
    {
        ...Error code...
        return false;                // If It Fails, Return False
    }
}
```

接下来我们将要看看它是否是RAW头。如果是，我们需要将此变量的值加1以获取紧随头之后的像素总数。

```
if(chunkheader < 128)          // 如果是RAW块
{
    chunkheader++;             // 变量值加1以获取RAW像素的总数
```

我们开启另一个循环读取所有的颜色信息。它将会循环块头中指定的次数，并且每次循环读取和存储一个像素。

首先，我们读取并检验像素数据。单个像素的数据将被存储在colorbuffer变量中。然后我们将检查它是否为RAW头。如果是，我们需要添加一个到变量之中以获取头之后的像素总数。

```
// 开始像素读取循环
for(short counter = 0; counter < chunkheader; counter++)
{
    // 尝试读取一个像素
    if(fread(colorbuffer, 1, tga.bytesPerPixel, fTGA) != tga.bytesPerPixel)
    {
        ...Error code...
        return false;           // 如果失败，返回false
    }
}
```

我们循环中的下一步将要获取存储在colorbuffer中的颜色值并且将其写入稍后将要使用的imageData变量中。在这个过程中，数据格式将会由BGR翻转为RGB或由BGRA转换为RGBA，具体情况取决于每像素的比特数。当我们完成任务后我们增加当前的字节和当前的像素计数器。

```
texture->imageData[currentbyte] = colorbuffer[2];      // 写“R”字节
texture->imageData[currentbyte + 1] = colorbuffer[1];  // 写“G”字节
texture->imageData[currentbyte + 2] = colorbuffer[0];  // 写“B”字节
if(tga.bytesPerPixel == 4)                            // 如果是32位图像...
{
    texture->imageData[currentbyte + 3] = colorbuffer[3]; // 写“A”字节
}
// 依据每像素的字节数增加字节计数器
currentbyte += tga.bytesPerPixel;
currentpixel++;           // 像素计数器加1
```

下一段处理描述RLE段的“块”头。首先我们将chunkheader减去127来得到获取下一个颜色重复的次数。

```
else          // 如果是RLE头
{
    chunkheader -= 127;           // 减去127获得ID Bit的Rid
```

然后我们尝试读取下一个像素值。

```
// 读取下一个像素  
if(fread(colorbuffer, 1, tga.bytesPerPixel, fTGA) != tga.bytesPerPixel)  
{  
    ...Error code...  
    return false;           // 如果失败 , 返回false  
}
```

接下来，我们开始循环拷贝我们多次读到内存中的像素，这由RLE头中的值规定。

然后，我们将颜色值拷贝到图像数据中，预处理R和B的值交换。

随后，我们增加当前的字节数、当前像素，这样我们再次写入值时可以处在正确的位置。

```
// 开始循环  
for(short counter = 0; counter < chunkheader; counter++)  
{  
    // 拷贝“ R ”字节  
    texture->imageData[currentbyte] = colorbuffer[2];  
    // 拷贝“ G ”字节  
    texture->imageData[currentbyte + 1] = colorbuffer[1];  
    // 拷贝“ B ”字节  
    texture->imageData[currentbyte + 2] = colorbuffer[0];  
    if(tga.bytesPerPixel == 4)      // 如果是32位图像  
    {  
        // 拷贝“ A ”字节  
        texture->imageData[currentbyte + 3] = colorbuffer[3];  
    }  
    currentbyte += tga.bytesPerPixel; // 增加字节计数器  
    currentpixel++;               // 增加字节计数器
```

只要仍剩有像素要读取，我们将会继续主循环。

最后，我们关闭文件并返回成功。

```
while(currentpixel < pixelcount); // 是否有更多的像素要读取？开始循环直到最后  
fclose(fTGA); // 关闭文件  
return true; // 返回成功  
}
```

现在你已经为glGenTextures和glBindTexture准备好了数据。我建议你查看Nehe的教程6和24以获取这些命令的更多信息。那证实了我先前写的教程的正确性，我不确保的代码中没有错误，虽然我努力使之不发生错误。特别感谢Jeff “ Nehe ” Molofee写了这个伟大的教程，以及Trent “ ShiningKnight ” Polack帮助我修订这个教程。如果你发现了错误、有建议或者注释，请自由地给我发Email

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

#### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。



## 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

## 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

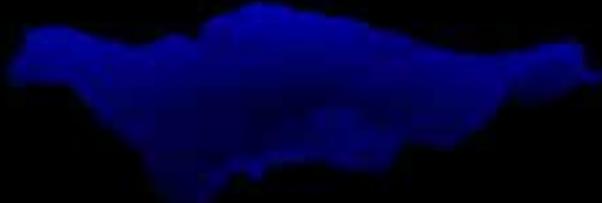
&lt; 第32课

第34课 &gt;



## 第34课

从高度图生成地形:



这一课将教会你如何从一个2D的灰度图创建地形

欢迎来到新的一课 , Ben Humphrey写了这一课的代码 , 它是基于第一课所写的。

在这一课里 , 我们将教会你如何使用地形 , 你将知道高度图这个概念。

下面我们来定义一些全局变量 , MAP\_SIZE是你使用的高度图的大小 , 在这一课里我们使用1024\*1024的地图。STEP\_SIZE设置高度图中相邻顶点之间的距离。HEIGHT\_RATIO设置在高度方向的缩放比例 , 越大地形看起来越陡峭。bRender设置使用多边形还是线绘制地形。

```
#define MAP_SIZE 1024
#define STEP_SIZE 16           // 相邻顶点的距离
#define HEIGHT_RATIO 1.5f
bool bRender = TRUE;        // true为多边形渲染 , false为线渲染
```

下面的代码用来保存高度数据

```
BYTE g_HeightMap[MAP_SIZE*MAP_SIZE];           // 保存高度数据  
float scaleValue = 0.15f;                      // 地形的缩放比例
```

下面的函数从文件中加载高度数据

```
// 从*.raw文件中加载高度数据  
void LoadRawFile(LPSTR strName, int nSize, BYTE *pHeightMap)  
{  
    FILE *pFile = NULL;  
  
    // 打开文件  
    pFile = fopen( strName, "rb" );  
  
    // 如果文件不能打开  
    if ( pFile == NULL )  
    {  
        // 提示错误，退出  
        MessageBox(NULL, "不能打开高度图文件", "错误", MB_OK);  
        return;  
    }  
  
    // 读取文件数据到pHeightMap数组中  
    fread( pHeightMap, 1, nSize, pFile );  
  
    // 读取是否成功  
    int result = ferror( pFile );  
  
    // 如果不成功，提示错误退出  
    if (result)  
    {  
        MessageBox(NULL, "读取数据失败", "错误", MB_OK);  
    }  
}
```

}

```
// 关闭文件  
fclose(pFile);
```

}

InitGL函数基本没有变化，只是加入了加载高度图的函数

```
// 载入1024*1024的高度图道g_HeightMap数组中
```

```
LoadRawFile("Data/Terrain.raw", MAP_SIZE * MAP_SIZE, g_HeightMap);
```

下面的函数返回(x,y)点的高度

```
int Height(BYTE *pHeightMap, int X, int Y) // 下面的函数返回(x,y)点的高度  
{  
    int x = X % MAP_SIZE; // 限制X的值在0-1024之间  
    int y = Y % MAP_SIZE; // 限制Y的值在0-1024之间  
  
    if(!pHeightMap) return 0; // 检测高度图是否存在，不存在则返回0
```

返回 (x,y)的高度

```
return pHeightMap[x + (y * MAP_SIZE)]; // 返回 (x,y)的高度  
}
```

按高度设置顶点的颜色，越高的地方越亮

```

void SetVertexColor(BYTE *pHeightMap, int x, int y)          // 按高度设置顶点的颜色，越高的地方越亮
{
    if(!pHeightMap) return;

    float fColor = -0.15f + (Height(pHeightMap, x, y ) / 256.0f);

    // 设置顶点的颜色
    glColor3f(0.0f, 0.0f, fColor );
}

```

下面的函数在OpenGL中，根据高度图渲染输出地形

```

void RenderHeightMap(BYTE pHeightMap[])          // 根据高度图渲染输出地形
{
    int X = 0, Y = 0;                            // 设置循环变量
    int x, y, z;

    if(!pHeightMap) return;                      // 确认高度图存在

    if(bRender)                                  // 选择渲染模式
        glBegin( GL_QUADS );                   // 渲染为四边形
    else
        glBegin( GL_LINES );                   // 渲染为直线

```

下面的函数求得每一点的坐标和颜色，调用OpenGL渲染

```

for ( X = 0; X < MAP_SIZE; X += STEP_SIZE )
    for ( Y = 0; Y < MAP_SIZE; Y += STEP_SIZE )
    {
        // 绘制(x,y)处的顶点

```

```
// 获得(x,y,z)坐标
x = X;
y = Height(pHeightMap, X, Y );
z = Y;

// 设置顶点颜色
SetVertexColor(pHeightMap, x, z);

glVertex3i(x, y, z);           // 调用OpenGL绘制顶点的命令

// 绘制(x,y+1)处的顶点
x = X;
y = Height(pHeightMap, X, Y + STEP_SIZE );
z = Y + STEP_SIZE ;
SetVertexColor(pHeightMap, x, z);
glVertex3i(x, y, z);

// 绘制(x+1,y+1)处的顶点
x = X + STEP_SIZE;
y = Height(pHeightMap, X + STEP_SIZE, Y + STEP_SIZE );
z = Y + STEP_SIZE ;
SetVertexColor(pHeightMap, x, z);
glVertex3i(x, y, z);

// 绘制(x+1,y)处的顶点
x = X + STEP_SIZE;
y = Height(pHeightMap, X + STEP_SIZE, Y );
z = Y;
SetVertexColor(pHeightMap, x, z);
glVertex3i(x, y, z);

}

glEnd();

glColor4f(1.0f, 1.0f, 1.0f, 1.0f);           // 重置颜色
}
```

DrawGLScene函数基本没变化，只是设置了视点和缩放系数，调用上面的函数绘制出地形。

```
// 设置视点  
gluLookAt(212, 60, 194, 186, 55, 171, 0, 1, 0);  
glScalef(scaleValue, scaleValue * HEIGHT_RATIO, scaleValue);  
  
RenderHeightMap(g_HeightMap); // 渲染高度图  
  
return TRUE;  
}
```

WndProc()函数基本没有变化，只是加入了单击左键的相应函数

```
case WM_LBUTTONDOWN: // 是否单击鼠标左键  
{  
    bRender = !bRender; // 改变渲染模式  
    return 0; // 返回  
}
```

上面就是所有绘制地形的代码了，简单吧。

希望你喜欢这个教程！

### 版权与使用声明：

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。



### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第33课

第35课 &gt;



## 第35课



在OpenGL中播放AVI:

在OpenGL中如何播放AVI呢？利用Windows的API把每一帧作为纹理绑定到OpenGL中，虽然很慢，但它的效果不错。你可以试试。

首先我得说我非常喜欢这一章节.Jonathan de Blok使我产生了用OpenGL编写AVI播放器的想法,可那时,我根本不知如何打开AVI文件,更不必说去写一个播放器了.于是我浏览了收藏的编程书籍,没有一本讲到AVI文件的.我又阅读了MSDN上和AVI文件格式有关的一切内容,上面有很多有用的信息,但我需要更多的.

花了几小时在网上搜到AVI范例,只找到两个网站.我的搜索技巧不能说很棒吧,但99.9%的情况,我能找到我要寻找的东西.了解到AVI范例竟如此之少时,我完全震惊了.大多数范例并不能编译通过...有一些则用了太复杂的的方法(至少对我如此),剩下的不错,可是用VB, Delphi等写的(不是用vc++).

找到的第一个网页是Jonathan Nix写的题为"AVI 文件"的文章.网址是<http://www.gamedev.net/reference/programming/features/avifile>.感谢Jonathan写了这片关于AVI格式的好文章.虽然我用不同的做法,但他的代码片断和清晰的注解让人学得很轻松!第二个网站标题为"AVI 总体观"(John F. McGowan, Ph.D写的)..我可以大肆赞美John的网页有多么惊奇,但你最好自己去看看.他的网址是<http://www.jmcgowan.com/avi.html>.这个网站讲到了和AVI格式有关的几乎所有内容.感谢John做了一个这么有用的网站.

最后要提到是我没有借鉴任何代码,没有抄袭任何代码.我的代码是花了三天时间了解到上述网站和文章的信息后才写成的.我是想说我的代码也许不是播放AVI文件的最好代码,他也许不是放AVI文件的正确代码,但他管用而且使用方便.如果你不喜欢这些代码和我的编程风格,或者觉得我的言论伤害到整个编程界,你有以下选择:1)在网上找到替换的资源2)写自己的AVI播放器3)写一篇更好的文章.任何访问本网站的人现在应该知道我只是一名中级程序员(这一点我在网站里很多文章的开头都提到过)!我编写代码自乐而已.本网站的目的在于让非精英程序员更轻松的开始OpenGL编程.这些文章只是关于我实现的几个特殊的效果...没有其他的.

开始讲代码首先你要注意的是我们要包括和连接到视频头文件和库文件.非常感谢微软(窝不敢相信我说了什么).库文件使打开,播放AVI文件都很简便.现在你要知道的是必须包括头文件vfw.h而且要连接到vfw32.lib库文件如果想编译你的代码的话:)

```
#include <vfw.h>                                // Video For Windows头文件
#include "NeHeGL.h"                               // NeHeGL头文件

#pragma comment( lib, "opengl32.lib" )
#pragma comment( lib, "glu32.lib" )
#pragma comment( lib, "vfw32.lib" )                // 链接到VFW32.lib

GL_Window* g_window;
Keys* g_keys;
```

现在定义变量.angle是用来根据时间来旋转物体的.为简单起见我们用angle来控制所有的旋转.

接下来是一个整型变量是用来计算经过的时间(以毫秒计).它使帧速保持一个速度.

后面细讲!

frame是动画要显示的当前帧,初始值为0(第一帧).我想如果成功打开AVI,他至少有一帧吧,这样假定比较安全:)

effect是当前屏幕上的效果(有:立方体,球体,圆柱体).env是布尔值.若它为true则环境映射启动,若为假,则物体没有环境映射.若bg为true,你会看到物体后有全屏的动画;若为假,你只会看到物体(没有背景).

sp,ep和bp用来确定使用者没有按着键不放.

```
float    angle;          // 旋转用
int     next;           // 动画用
int     frame=0;        // 帧计数器
```

```

int      effect;           // 当前效果
bool     sp;               // 空格键按下?
bool     env=TRUE;         // 环境映射(默认开)
bool     ep;               // 'E' 按下?
bool     bg=TRUE;          // 背景(默认开)
bool     bp;               // 'B' 按下?

```

psi结构体包含AVI文件信息.pavi缓冲的指针,缓冲用来接受AVI文件打开时的流句柄.pgf是指向GetFrame对象的指针.bmih在后面的代码中将被用来把动画的每一帧转换为我们需要的格式(保存位图的头信息).lastframe保存AVI动画最后一帧的序号.width和height保存AVI流的维信息,最后...pdata是图象数据的指针(每次在从AVI中获得一帧后返回).mpf用来计算每帧需要多少毫秒.后面细谈这个变量.

```

AVISTREAMINFO    psi;           // 包含流信息的结构体的指针
PAVISTREAM       pavi;          // 流句柄
PGETFRAME        pgf;           // GetFrame对象的指针
BITMAPINFOHEADER  bmih;          // 头信息 For DrawDibDraw
long              lastframe;     // 流中最后一帧
int               width;          // 视频宽
int               height;         // 视频高
char              *pdata;          // 纹理数据指针
int               mpf;            // 控制每帧显示时间

```

在本章中我们用GLU库创建两个二次曲面(球体和圆柱体).quadratic是曲面对象的指针.hdd是DrawDib设备上下文的句柄.hdc是设备上下文的句柄.hBitmap是设备无关位图的句柄(在后面位图转换时用到).data是最指向转换后位图的图象数据的指针,在后面的代码中会有意义,往下读:)

```

GLUquadricObj *quadratic;      // 存储二次曲面对象

HDRAWDIB hdd;                 // Dib句柄
HBITMAP hBitmap;               // 设备无关位图的句柄
HDC hdc = CreateCompatibleDC(0); // 创建一个兼容的设备上下文
unsigned char* data = 0;         // 调整后的图象数据指针

```

下面使用到汇编语言.那些从来没有用过汇编的不要被吓倒了.他看起来神秘,实际上非常简单!

在写本章是我发现了十分奇怪的事.第一次做出来的可以播放,但色彩混乱了.本来是红色的变成蓝色的了,本来是蓝色的变成红色的了.我简直要发狂了!我相信我的代码某处有问题.看了一边代码还是找不到bug于是又读了MSDN.为什么红色与蓝色互换了!?!MSDN明明说24比特位图是RGB啊!又读了一些东西,我找到了答案.在WINDOWS图形系统中,RGB数据是倒着存储的(BGR).而在OpenGL中,要用的RGB数据就是RGB的顺序!

在抱怨了微软之后:)我决定加一条注解!我不因为RGB数据倒过来存放而打算骂微软.只是觉得很奇怪--他叫做RGB实际上在文件中是按BGR存的!

另:这一点和"little endian"和"big endian"有关.Intel以及Intel兼容产品用little endian--LSB(数据最低位)首先存.OpenGL是产生于Silicon Graphics的机器的,用的是big endian,所以标准的OpenGL位图格式是big endian格式.这是我的理解.

棒极了!所以说这第一个播放器就是一个垃圾!我的解决方法是用一个循环把数据交换过来.这能行,但太慢.我又在纹理生成代码中用GL\_BGR\_EXT代替了GL\_RGB,速度暴升,色彩显示也对了!问题解决了...原来我是这样想!后来发现一些OpenGL驱动不支持GL\_BGR...:(

与好友Maxwell Sayles讨论后,他推荐我用汇编代码来交换数据.一分钟后,他用icq发来下面的代码!也许不是最优化的,但他很快也很有效!

动画的每一帧存在一个缓冲里.图象256像素宽,256像素高,每个色彩一字节(一像素3字节).下面的代码会扫描整个缓冲并交换红与蓝的字节.红存在ebx+0,蓝存在ebx+2.我们一次向前走3字节(因为一个像素3字节).不断扫描直到所有数据交换过来.

你们有些人不喜欢用汇编代码,所以我想有必要在本章里解释一下.本来计划用GL\_BGR\_EXT,他管用,但不是所有的显卡都支持!我又用异或交换法,这在所有机器上都是有效的,但不十分快.用了汇编后速度相当快.考虑到我们在处理实时视频,你需要最快的交换方法.权衡了以上选择,汇编是最好的!如果你有更好的办法,就用你自己的吧!我并不是告诉你必须如何去做,只是告诉你我的做法.我也会细致的解释代码.如果你要用更好的代码来作替换,你要清楚这些代码是来干什么的,自己写代码时,要为日后的优化提供方便.

```
void flip(void* buffer) // 交换红蓝数据(256x256)
{
    void* b = buffer; // 缓冲指针
    __asm // 汇编代码
```

```

{
    mov ecx, 256*256          // 设置计数器
    mov ebx, b                 // ebx存数据指针
    label:
        mov al,[ebx+0]          // 把ebx位置的值赋予al
        mov ah,[ebx+2]          // 把ebx+2位置的值赋予ah
        mov [ebx+2],al           // 把al的值存到ebx+2的位置
        mov [ebx+0],ah           // 把ah的值存到ebx+0的位置

        add ebx,3               // 向前走3个字节
        dec ecx                // 循环计数器减1
        jnz label               // ecx非0则跳至label
    }
}

```

下面的代码以只读方式打开AVI文件.szFile是打开文件的名字.title[100]用来修改window标题(显示AVI文件信息).

首先调用AVIFileInit().他初始化AVI文件库(使东西能用?鹄?).

打开AVI文件有很多方法.我采用AVIStreamOpenFromFile(...).他能打开AVI文件中单独一个流(AVI文件可以包含多个流).它的参数如下:pavi是接收流句柄的缓冲的指针,szFile是打开文件的名字(包括路径).第三参数是打开的流的类型.在这个工程里,我们只对视频流感兴趣(streamtypeVIDEO).第四参数是0,这表示我们需要第一次读到的视频流(一个AVI文件里会有多个视频流,我们要第一个).OF\_READ表示以只读方式打开文件.最后一个参数是一个类标识句柄的指针.说实话,我也不清楚他是干吗的.我让windows自己设定,于是把NULL传过去.

```

void OpenAVI(LPCSTR szFile)                      // 打开AVI文件szFile
{
    TCHAR title[100];                            // 包含修改了的window标题
    AVIFileInit();                             // 打开AVI文件库

    // 打开AVI流
    if (AVIStreamOpenFromFile(&pavi, szFile, streamtypeVIDEO, 0, OF_READ, NULL) !=0)
    {
        // 打开流时的出错处理
        MessageBox (HWND_DESKTOP, "打开AVI流失败", "错误", MB_OK |

```

```
MB_ICONEXCLAMATION);  
}
```

到目前为止,我们假定文件被正确打开,流被正确定位!然后用AVIStreamInfo(...)从AVI文件里抓取一些信息.

先前我们创建了叫psi的结构体来保存AVI流的信息.下面第一行,我们把AVI信息填入该结构体.从流的宽度(以像素计)到动画的帧速等所有的信息都会存到psi中.那些想要精确控制播放速度的要记住我刚才说的.更多的信息参阅MSDN.

我们通过右边位置减左边位置算出帧宽.这个结果是以像素记的精确的帧宽.至于高度,可以用底边位置减顶边位置得到.这样得到高度的像素值.

然后用AVIStreamLength(...)得到AVI文件最后一帧的序号.AVIStreamLength(...)返回动画最后一帧的序号.结果存在lastframe里.

计算帧速很简单.每秒帧速(fps)= psi.dwRate/psi.dwScale.返回的值应该匹配显示帧的速度(你在AVI动画中右击鼠标可以看到).你会问那么这和mpf有什么关系呢?第一次写这个代码时,我试着用fps来选择动画了正确的帧面.我遇到一个问题...视频放的太快!于是我看了下视频属性.face2.avi文件有3.36秒长.帧速是29.974fps.视频动画共有91帧.而 $3.36 * 29.974 = 100.71$ .非常奇怪!!

所以我采用一些不同的方法.不是计算帧速,我计算每一帧播放所需时间.

AVIStreamSampleToTime()把在动画中的位置转换位你到达该位置所需的时间(毫秒计).所以通过计算到达最后一帧的时间就得到整个动画的播放时间.再拿这个结果除以动画总帧数(lastframe).这样就给出了每帧的显示时间(毫秒计).结果存在mpf(milliseconds per frame)里.你也能通过获取动画中一帧的时间来算每帧的毫秒数,代码为:AVIStreamSampleToTime(pavi,1).两种方法都不错!非常感谢Albert Chaulk提供思路!

我说每帧的毫秒数不精确是因为mpf是一个整型值,所以所有的浮点数都会被取整.

```
AVIStreamInfo(pavi, &psi, sizeof(psi));           // 把流信息读进psi  
width=psi.rcFrame.right-psi.rcFrame.left;        // 宽度为右边减左边  
height=psi.rcFrame.bottom-psi.rcFrame.top;         // 高为底边减顶边  
  
lastframe=AVIStreamLength(pavi);                   // 最后一帧的序号  
  
mpf=AVIStreamSampleToTime(pavi,lastframe)/lastframe; // mpf的不精确值
```

因为OpenGL需要纹理数据是2的幂,而大多视频是160\*120,320\*240等等,所以需要一种把视频格式重调整为能用作纹理的格式.我们可利用Windows Dib函数去做.

首先要做的是描述我们想要的图像的类型.于是我们要以所需参数填好bmih这个BitmapInfoHeader结构.

首先设定该结构体的大小.再把位平面数设为1.3字节的数据有24比特(RGB).要使图像位256像素宽,256像素高,最后要让数据返回为UNCOMPRESSED(非压缩)的RGB数据(BI\_RGB).

CreateDIBSection创建一个可直接写的设备无关位图(dib).如果一切顺利,hBitmap会指向该dib的比特值.hdc是设备上下文(DC)的句柄第二参数是BitmapInfo结构体的指针.该结构体包含了上述dib文件的信息.第三参数(DIB\_RGB\_COLORS)设定数据是RGB值.data是指向DIB比特值位置的指针的指针(呜,真绕口).第五参数设为NULL,我们的DIB已被分配好内存.末了,最后一个参数可忽略(设为NULL).

引自MSDN:SelectObject函数选一个对象进入设备上下文(DC).

现在我们建好一个能直接写的DIB, yeah:)

```
bmih.biSize      = sizeof (BITMAPINFOHEADER);      // BitmapInfoHeader的大小  
bmih.biPlanes   = 1;                            // 位平面  
bmih.biBitCount = 24;                           // 比特格式(24 Bit, 3 Bytes)  
bmih.biWidth    = 256;                           // 宽度(256 Pixels)  
bmih.biHeight   = 256;                           // 高度 (256 Pixels)  
bmih.biCompression = BI_RGB;                    // 申请的模式 = RGB
```

```
hBitmap = CreateDIBSection (hdc, (BITMAPINFO*)(&bmih), DIB_RGB_COLORS, (void**)  
(&data), NULL, NULL);  
SelectObject (hdc, hBitmap);                      // 选hBitmap进入设备上下文(hdc)
```

在从AVI中读取帧面前还有几件事要做.接下来使程序做好从AVI文件中解出帧面的准备.  
用AVIStreamGetFrameOpen(...)函数做这一点.

你能给这个函数传一个结构体作为第二参数(它会返回一个特定的视频格式).糟糕的是,你能改变的唯一数据是返回的图像的宽度和高度.MSDN也提到能传AVIGETFRAMEF\_BESTDISPLAYFMT为参数来选择一个最佳显示格式.奇怪的是,我的编译器没有定义这玩艺儿.

如果一切顺利,一个GETFRAME对象被返回(用来读帧数据).有问题的话,提示框会出现在屏幕上告诉你有错误!

```
pgf=AVIStreamGetFrameOpen(pavi, NULL);           // 用要求的模式建PGETFRAME
if (pgf==NULL)
{
    // 解帧出错
    MessageBox (HWND_DESKTOP, "不能打开AVI帧", "错误", MB_OK | MB_ICONEXCLAMATION);
}
```

下面的代码把视频宽,高和帧数传给window标题.用函数SetWindowText(...)在window顶部显示标题.以窗口模式运行程序看看以下代码的作用.

```
// bt标题栏信息(宽 / 高/ 帧数)
wsprintf (title, "NeHe's AVI Player: Width: %d, Height: %d, Frames: %d", width, height, lastframe);
SetWindowText(g_window->hWnd, title);           // 修改标题栏
}
```

下面是有意思的东西...从AVI中抓取一帧,把它转为大小和色深可用的图象.lpbi包含一帧的BitmapInfoHeader信息.我们在下面第二行完成了几件事.先是抓了动画的一帧...我们需要的帧面由这些帧确定.这会让动画走掉这一帧,lpbi会指向这一帧的头信息.

下面是有趣的东西...我们要指向图像数据了.要跳过头信息(lpbi->biSize).一件事直到写本文时我才意识到:也要跳过任何的色彩信息.所以要跳过biClrUsed\*sizeof(RGBQUAD)(译者:我想他是说要跳过调色板信息).做完这一切,我们就得到图像数据的指针了(pdata).

也要把动画的每一帧的大小转为纹理能用的大小,还要把数据转为RGB数据.这用到DrawDibDraw(...).

一个大概的解释.我们能直接写设定的DIB图像.那就是DrawDibDraw(...)所做的.第一参数是DrawDib DC的句柄.第二参数是DC的句柄.接下来用左上角(0,0)和右下角(256,256)构成目标矩形.

lpbi指向刚读的帧的bitmapinfoheader信息 pdata是刚读的帧的图像数据指针.

再把源图象(刚读的帧)的左上角设为(0,0),右下角设为(帧宽,帧高).最后的参数应设为0.

这个方法可把任何大小、色深的图像转为256\*256\*24bit的图像.

```
void GrabAVIFrame(int frame) // 从流中抓取一帧
{
    LPBITMAPINFOHEADER lpbi; // 存位图的头信息
    lpbi = (LPBITMAPINFOHEADER)AVIStreamGetFrame(pgf, frame); // 从AVI流中得到数据
    pdata=(char *)lpbi+lpbi->biSize+lpbi->biClrUsed * sizeof(RGBQUAD); // 数据指针,由
AVIStreamGetFrame返回(跳过头
//信息和色彩信息)
// 把数据转为所需格式
    DrawDibDraw (hdd, hdc, 0, 0, 256, 256, lpbi, pdata, 0, 0, width, height, 0);
```

我们得到动画的每帧数据(红蓝数据颠倒的).为解决这个问题,我们的高速代码`flipIt(...)`.记住,data是指向DIB比特值位置的指针的指针变量.这意味着调用DrawDibDraw后,data指向一个调整过大小(256\*256),修改过色深(24bits)的位图数据.

原来我通过重建动画的每一帧来更新纹理.我收到几封email建议我用`glTexSubImage2D()`.翻阅了OpenGL红宝书后,我磕磕绊绊的写出下面注释:"创建纹理的计算消耗比修改纹理要大.在OpenGL1.1版本中,有几条调用能更新全部或部分纹理图像信息.这对某些应用程序有用,比如实时的抓取视频图像作纹理.对于这些程序,用`glTexSubImage2D()`根据新视频图像来创建单个纹理以代替旧的纹理数据是行得通的."

在我个人并没有发现速度明显加快,也许在低端显卡上才会.`glTexSubImage2D()`的参数是:目标是一个二维纹理(GL\_TEXTURE\_2D).细节级别(0),mipmapping用.x(0),y(0)告诉OpenGL开始拷贝的位置(0,0是纹理的左下角).然后是图像的宽度,我们要拷贝的图像是256像素宽,256像素高,GL\_RGB是我们的数据格式.我们在拷贝无符号byte.最后...图像数据指针---data.非常简单!

Kevin Rogers 另加:我想指出使用`glTexSubImage2D()`另一个重要原因.不仅因为在许多OpenGL实现中它很快,还因为目标区不必是2的幂.这对视频重放很方便,因为一帧的维通常不是2的幂(而是像320\*200之类的).这样给了你很大机动性,你可以按视频流原本的样子播放,而不是扭曲或剪切每一帧来适应纹理的维.

重要的是你不能更新一个纹理如果你第一次没有创建他!我们在Initialize()中创建纹理.

还要提到的是...如果你计划在工程里使用多个纹理,务必绑住你要更新的纹理.否则,更新出来的纹理也许不是你想要的!

```

    flipIt(data); // 交换红蓝数据
    // 更新纹理
    glTexSubImage2D (GL_TEXTURE_2D, 0, 0, 0, 256, 256, GL_RGB, GL_UNSIGNED_BYTE, data);
}

```

接下来的部分当程序退出时调用,我们关掉DrawDib DC,释放占用的资源.然后释放AVI GetFrame资源.最后释放AVI流和文件.

```

void CloseAVI(void) // 关掉AVI资源

```

```
{
    DeleteObject(hBitmap);           //释放设备无关位图信息
    DrawDibClose(hdd);              // 关掉DrawDib DC
    AVIStreamGetFrameClose(pgf);    // 释放AVI GetFrame资源
    AVIStreamRelease(pavi);         // 释放AVI流
    AVIFileExit();                 // 释放AVI文件
}
```

初始化很简明.设初始的angle为0.再打开DrawDib库(得到一个DC).一切顺利的话,hdd会是新创建的dc的句柄.

以黑色清屏,开启深度测试,等等.

然后建一个新的二次曲面.quadratic是这个新对象的指针.设置光滑的法线,允许纹理坐标生成.

```
BOOL Initialize (GL_Window* window, Keys* keys)
{
    g_window = window;
    g_keys = keys;

    // 开始用户的初始
    angle = 0.0f;                      // angle为0先
    hdd = DrawDibOpen();                // 得到Dib的DC
    glClearColor (0.0f, 0.0f, 0.0f, 0.5f); // 黑色背景
    glClearDepth (1.0f);                // 深度缓冲初始
    glDepthFunc (GL_LESS);             // 深度测试的类型(小于或等于)
    glEnable(GL_DEPTH_TEST);            // 开启深度测试
    glShadeModel (GL_SMOOTH);          // 平滑效果
    glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // 透视图计算设为 //最高精度

    quadratic=gluNewQuadric();          // 建二次曲面的指针
    gluQuadricNormals(quadratic, GLU_SMOOTH); // 设置光滑的法线
    gluQuadricTexture(quadratic, GL_TRUE); // 创建纹理坐标
```

下面的代码中,我们开启2D纹理映射,纹理滤镜设为GLNEAREST(最快,但看起来很糙),建立球面映射(为了实现环境映射效果).试试其它滤镜,如果你有条件,可以试试GLLINEAR得到一个平滑的动画效果.

设完纹理和球面映射,我们打开.AVI文件.我尽量使事情简单化...你能看出来么:)我们要打开的文件叫作facec2.avi

```

glEnable(GL_TEXTURE_2D);           // 开启2D纹理映射
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST); // 设置纹理滤
镜
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);

glTexGeni(GL_S,GL_TEXTURE_GEN_MODE,GL_SPHERE_MAP);      // 设纹理坐标生成模式
为s
glTexGeni(GL_T,GL_TEXTURE_GEN_MODE,GL_SPHERE_MAP);      // 设纹理坐标生成模
式为t

OpenAVI("data/face2.avi");          // 打开AVI文件

// 创建纹理
glTexImage2D(GL_TEXTURE_2D,0,GL_RGB,256,256,0,GL_RGB,GL_UNSIGNED_BYTE,data);

return TRUE;                      // 初始化成功返回TRUE
}

```

关闭时调用CloseAVI().他正确的关闭AVI文件,并释放所有占用资源.

```

void Deinitialize (void)           // 做所有的释放工作
{
    CloseAVI();                  // 关闭AVI文件
}

```

到了检查按键和更新旋转角度的地方了.我知道再没有必要详细解释这些代码了.我们检查空格键是否按下,若是,则增加effect值.有3种效果(立方,球,圆柱)第四个效果被选时(effect = 3)不画任何对象...仅显示背景!如果选了第四效果,空格又按下了,就重设为第一个效果(effect = 0).Yeah,我本该叫他对象:)

然后检查 ' b ' 键是否按下,若是,则改变背景(bg从ON到OFF或从OFF到ON).

环境映射的键设置也一样.检查 ' E ' 是否按下,若是则改变env从TRUE到FALSE或从FALSE到TRUE.仅仅是关闭或开启环境映射!

每次调用Update()时angle都加上一个小分数.我用经过的时间除以60.0f使速度降一点.

```
void Update (DWORD milliseconds) // 动画更新
{
    if (g_keys->keyDown [VK_ESCAPE] == TRUE) //ESC按下?
    {
        TerminateApplication (g_window); // 关闭程序
    }

    if (g_keys->keyDown [VK_F1] == TRUE) // F1按下?
    {
        ToggleFullscreen (g_window); // 改变显示模式
    }

    if ((g_keys->keyDown [' ']) && !sp) // 空格按下并已松开
    {
        sp=TRUE; // 设sp为True
        effect++; // 增加effect
        if (effect>3) // 超出界限?
            effect=0; // 重设为0
    }

    if (!g_keys->keyDown[' ']) // 空格没按下?
        sp=FALSE; // 设sp为False

    if ((g_keys->keyDown ['B']) && !bp) // ' B ' 按下并已松开
    {
        bp=TRUE; // 设bp为True
        bg=!bg; // 改变背景 Off/On
    }
}
```

```

if (!g_keys->keyDown['B'])           // 'B' 没按下?
    bp=FALSE;                      // 设bp为False

if ((g_keys->keyDown ['E']) && !ep)      // 'E' 按下并已松开
{
    ep=TRUE;                      // 设ep为True
    env=!env;                     // 改变环境映射 Off/On
}

if (!g_keys->keyDown['E'])           // 'E' 没按下
    ep=FALSE;                      // 设ep为False

angle += (float)(milliseconds) / 60.0f; // 根据时间更新angle

```

在原来的文章里,所有的AVI文件都以相同的速度播放.于是,我重写了本文让视频以正常的速度播放.next增加经过的毫秒数.如果你记得文章的前面,我们算出了显示每帧的毫秒数(mpf).为了计算当前帧,我们拿经过的时间除以显示每帧的毫秒数(mpf).

还要检查确定当前帧没有超过视频的最后一帧.若超过了,则将frame设为0,动画计时器设为0,于是动画从头开始.

下面的代码会丢掉一些帧,若果你的计算机太慢的话,  
或者另一个程序占用了CPU.如果想显示每一帧而不管计算机有多慢的话,你要检查next是否比mpf大,若是,你要把next设为0,frame增1.两种方法都行,虽然下面的代码更有利与跑的快的机器.

如果你有干劲,试着加上循环,快速播放,暂停或倒放等功能.

```

next+= milliseconds;           // 根据时间增加next
frame=next/mpf;               // 计算当前帧号

if (frame>=lastframe)         // 超过最后一帧?
{
    frame=0;                  // Frame设为0
    next=0;                   // 重设动画计时器
}
}

```

下面是画屏代码:)我们清屏和深度缓冲.再抓取动画的一帧.我将使这更简单!把你想要的帧数传给GrabAVIFrame().非常简单!当然,如果是多个AVI,你要传一个纹理标号.(你要做更多的事)

```
void Draw (void)                                // 绘制我们的屏幕
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清屏和深度缓冲
    GrabAVIFrame(frame);                            // 抓取动画的一帧
```

下面检查我们是否想画一个背景图.若bg是TRUE,重设模型视角矩阵,画一个单纹理映射的能盖住整个屏幕的矩形(纹理是从AVI从得到的一帧).矩形距离屏面向里20个单位,这样它看起来在对象之后(距离更远).

```
if (bg)                                         // 背景可见?
{
    glLoadIdentity();                           // 重设模型视角矩阵
    glBegin(GL_QUADS);                        // 开始画背景(一个矩形)
    // 正面
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 11.0f, 8.3f, -20.0f);
    glTexCoord2f(0.0f, 1.0f); glVertex3f(-11.0f, 8.3f, -20.0f);
    glTexCoord2f(0.0f, 0.0f); glVertex3f(-11.0f, -8.3f, -20.0f);
    glTexCoord2f(1.0f, 0.0f); glVertex3f( 11.0f, -8.3f, -20.0f);
    glEnd();
}
```

画完背景(或没有),重设模型视角矩阵(使视角中心回到屏幕中央).视角中心再向屏内移进10个单位.然后检查env是否为TRUE.若是,开启球面映射来实现环境映射效果.

```
glLoadIdentity();                           // 重设模型视角矩阵
glTranslatef (0.0f, 0.0f, -10.0f);        // 视角中心再向屏内移进10个单位
```

```

if (env) // 环境映射开启?
{
    glEnable(GL_TEXTURE_GEN_S); // 开启纹理坐标生成S坐标
    glEnable(GL_TEXTURE_GEN_T); // 开启纹理坐标生成T坐标
}

```

在最后关头我加了以下代码.他绕X轴和Y轴旋转(根据angle的值)然后在Z轴方向移动2单位.这使我们离开了屏幕中心.如果删掉下面三行,对象会在屏幕中心打转.有了下面三行,对象旋转时看起来离我们远一些:)

如果你不懂旋转和平移...你就不该读这一章:)

```

glRotatef(angle*2.3f,1.0f,0.0f,0.0f); // 加旋转让东西动起来
glRotatef(angle*1.8f,0.0f,1.0f,0.0f); // 加旋转让东西动起来
glTranslatef(0.0f,0.0f,2.0f); // 旋转后平移到新位置

```

下面的代码检查我们要画哪一個对象.若effect为0,我们做一些旋转在画一个立方体.这个旋转使立方体绕X,Y,Z轴旋转.现在你脑中该烙下建一个立方体的方法了吧:)

```

switch (effect) // 哪个效果?
{
case 0: // 效果 0 - 立方体
    glRotatef (angle*1.3f, 1.0f, 0.0f, 0.0f);
    glRotatef (angle*1.1f, 0.0f, 1.0f, 0.0f);
    glRotatef (angle*1.2f, 0.0f, 0.0f, 1.0f);
    glBegin(GL_QUADS);
        glNormal3f( 0.0f, 0.0f, 0.5f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);

        glNormal3f( 0.0f, 0.0f, -0.5f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);
}

```

```

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

glNormal3f( 0.0f, 0.5f, 0.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, -1.0f);

glNormal3f( 0.0f,-0.5f, 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);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 1.0f);

glNormal3f( 0.5f, 0.0f, 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);
glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 1.0f);

glNormal3f(-0.5f, 0.0f, 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);
glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 1.0f);
glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, -1.0f);

glEnd();
break;

```

下面是画球体的地方.开始先绕X,Y,Z轴旋转,再画球体.球体半径为1.3f,20经线,20纬线.我用20是因为我没打算让球体非常光滑.少用些经纬数,使球看起来不那么光滑,这样球转起来时就能看到球面映射的效果(当然球面映射必须开启).试着尝试其它值!要知道,使用更多的经纬数需要更强的计算能力!

case 1: // 效果1 , 球体

```

glRotatef (angle*1.3f, 1.0f, 0.0f, 0.0f);
glRotatef (angle*1.1f, 0.0f, 1.0f, 0.0f);
glRotatef (angle*1.2f, 0.0f, 0.0f, 1.0f);
gluSphere(quadratic,1.3f,20,20);
break;

```

下面画圆柱.开始先绕X,Y,Z轴旋转,圆柱顶和底的半径都为1.0f,高3.0f,32经线,32纬线.若减少经纬数,圆柱的组成多边形会减少,他看起来就没那么圆.

```
case 2: // 效果2 , 圆柱
    glRotatef (angle*1.3f, 1.0f, 0.0f, 0.0f);
    glRotatef (angle*1.1f, 0.0f, 1.0f, 0.0f);
    glRotatef (angle*1.2f, 0.0f, 0.0f, 1.0f);
    glTranslatef(0.0f,0.0f,-1.5f);
    gluCylinder(quadratic,1.0f,1.0f,3.0f,32,32);
    break;
}
```

下面检查env是否为TRUE,若是,关闭球面映射.调用glFlush()清空渲染流水线(使在下一帧开始前一切都渲染了).

```
if (env) // 是否开启了环境渲染
{
    glDisable(GL_TEXTURE_GEN_S); // 关闭纹理坐标S
    glDisable(GL_TEXTURE_GEN_T); // 关闭纹理坐标T
}

glFlush (); // 清空渲染流水线
}
```

希望你们喜欢这一章.现在已经凌晨两点了(译者oak:译到这时刚好也是2:00am!)...写这章花了我6小时了.听起来不可思议,可要把东西写通不是件容易的事.本文我读了三遍,我力图使文章好懂.不管你信还是不信,对我最重要的是你们能明白代码是怎样运作的,它为什么能行.那就是我喋喋不休并且加了过量注解的原因.

无论如何,我都想听到本文的反馈.如果你找到文章的错误,并想帮我做一些改进,请联系我.就像我说的那样,这是我第一次写和AVI有关的代码.通常我不会写一个我才接触到的主题,但我太兴奋了,并且考虑到关于这方面的文章太少了.我所希望的是,我打开了编写高质量AVI demo和代码的一扇门!也许成功,也许没有.不管怎样,你可以任意处理我的代码.

非常感谢 Fredster提供face AVI文件.Face是他发来的六个AVI动画中的一个.他没提出任何问题和条件.他以他的方式帮助了我,谢谢他!

更要感谢Jonathan de Blok,要没要她,本文就不会有.他给我发来他的AVI播放器的代码,这使我对AVI格式产生了兴趣.他也回答了我问的关于他的代码的问题.但重要的是我并没有借鉴或抄袭他的代码,他的代码只是帮助我理解AVI播放器的运行机制.我的播放器的打开,解帧和播放AVI文件用的是不同的代码!

感谢给予帮助的所有人,包括所有参观者!若没有你们,我的网站不值一文!!!

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

#### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,



任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

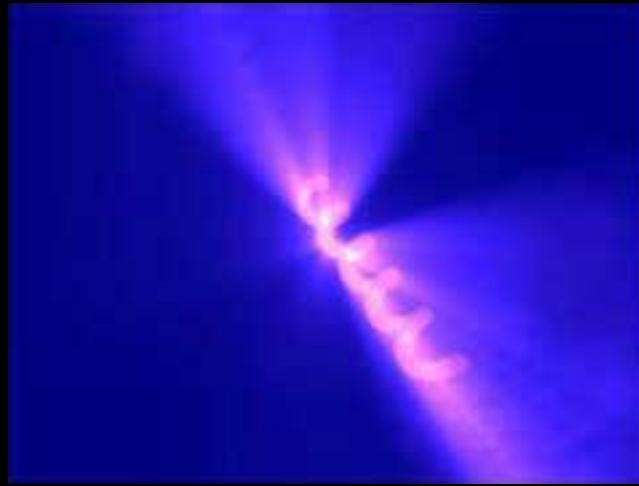
感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第34课

第36课 >



## 第36课



放射模糊和渲染到纹理:

如何实现放射状的滤镜效果呢，看上去很难，其实很简单。把渲染得图像作为纹理提取出来，在利用OpenGL本身自带的纹理过滤，就能实现这种效果，不信，你试试。

嗨，我是Dario Corno,也因SpinningKids的rIo而为大家所知。首先，我想要解释我为什么决定写这点指南。我自1989年以来就从事scener的工作。我想要你们去下载一些demo(示例程序，也就是演示——译者)以帮助你理解什么是Demo并且demo的效果是什么。

Demos是被用来展示恰似风雅的技术一样无限并且时而严酷的译码。在今天的演示中你通常总可以发现一些真正迷人的效果。这不是一本迷人的效果指南，但结果将非常的酷！你能够从<http://www.pouet.net>和<http://ftp.scene.org>. 发现大量的演示收集。

既然绪论超出了我们探讨的范围，我们可以继续我们的指南了。

我将解释如何做一个看起来象径向模糊的eye candy 效果。有时它以测定体积的光线被提到。不要相信，它仅仅是一个冒牌的辐射状模糊;D

辐射状模糊效果通常借助于模糊在一个方向上相对于模糊物的中心原始图象的每一个象素来做的。

借助于现今的硬件用色彩缓冲器来手工作模糊处理是极其困难的（至少在某种程度上它被所有的gfx卡所支持），因此我们需要一些窍门来达到同样的效果。

作为一个奖励当学习径向模糊效果时，你同样将学到如何轻松地提供材料的纹理。

我决定在这篇指南中使用弹簧作为外形因为它是一个酷的外形，另外还因为我对立方体感到厌烦：}

多留意这篇指南关于如何创建那个效果的指导方针是重要的。我不研究解释那些代码的

详情。你应当用心记下它们中的大部分 : }

下面是变量的定义和用到的头文件。

```
#include <math.h> // 数学库

float angle; // 用来旋转那个螺旋
float vertexes[3][3]; // 为3个设置的顶点保存浮点信息
float normal[3]; // 存放法线数据的数组
GLuint BlurTexture; // 存放纹理编号的一个无符号整型
```

函数EmptyTexture()创建了一个空的纹理并返回纹理的编号。我们刚分配了一些自由空间（准确的是128\*128\*4无符号整数）。

128\*128是纹理的大小（128象素宽和高），4意味着为每一个象素我们想用4byte来存储红，绿，蓝和ALPHA组件。

```
GLuint EmptyTexture() // 创建一个空的纹理
{
    GLuint txtnumber; // 纹理ID
    unsigned int* data; // 存储数据

    // 为纹理数据 (128*128*4) 建立存储区
    data = (unsigned int*)new GLuint[((128 * 128) * 4 * sizeof(unsigned int))];
```

在分配完空间之后我们用ZeroMemory函数清0，返回指针（数据）和被清0的存储区的大小。

另一半需注意的重要的事情是我们设置GL\_LINEAR的放大率和缩放率的方法。因为我们  
将被我们的纹理要求投入全部的精力并且如果被滥用，GL\_NEAREST会看起来非常糟糕。

```
ZeroMemory(data,((128 * 128) * 4 * sizeof(unsigned int))); // 清除存储区

glGenTextures(1, &txtnumber); // 创建一个纹理
glBindTexture(GL_TEXTURE_2D, txtnumber); // 构造纹理
glTexImage2D(GL_TEXTURE_2D, 0, 4, 128, 128, 0,
```

```

    GL_RGBA, GL_UNSIGNED_BYTE, data);           // 用数据中的信息构造纹理
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);

delete [] data;                           // 释放数据

return txtnumber;                         // 返回纹理ID
}

```

这个函数简单规格化法线向量的长度。向量被当作有3个浮点类型的元素的数组来表示，第一个元素表示X轴，第二个表示Y，第三个表示Z。一个规格化的向量[Nv]被Vn表达为 $V_n = [V_{ox}/|V_o|, V_{oy}/|V_o|, V_{oz}/|V_o|]$ ，这里 $V_o$ 是最初的向量， $|V_o|$ 是该向量的系数（或长度），X，Y，Z它的组件。之后由向量的长度区分每一个法线向量组件。

```

void ReduceToUnit(float vector[3])           // 归一化一个法向量
{
    float length;                          // 一定长度的单位法线向量
    // 计算向量
    length = (float)sqrt((vector[0]*vector[0]) + (vector[1]*vector[1]) + (vector[2]*vector[2]));

    if(length == 0.0f)                     // 避免除0错误
        length = 1.0f;                    // 如果为0设置为1

    vector[0] /= length;                  // 归一化向量
    vector[1] /= length;
    vector[2] /= length;
}

```

下面各项计算所给的3个顶点向量（总在3个浮点数组中）。我们有两个参数：v[3][3]和out[3]。当然第一个参数是一个m=3，n=3每一行代表三角形一个顶点的浮点矩阵。Out是我们要放置作为结果的法线向量的位置。

相当简单的数学。我们将使用著名的交叉乘积运算。理论上说交叉乘积是两个向量——它返回另一个直交向量到两个原始向量——之间的操作。法线向量是一个垂直物体表面的直交向量，是与该表面相对的（通常一个规格化的长度）。设想两个向量是在一个三角形的一侧的上方，那么这个三角形两边的直交向量（由交叉乘积计算）就是那个三角形的法线。

解释比实行还难。

我们将着手从现存的顶点0到顶点1，从顶点1到顶点2找到那个向量。这是基本上通过减法——下一个顶点的每个组件减一个顶点的每个组件——作好了的。现在我们已经为我们的三角形的边找到了那个向量。通过交叉相乘我们为那个三角形找到了法线向量。看代码。

v[0][]是第一个顶点，v[1][]是第二个顶点，v[2][]是第三个顶点。每个顶点包括：v[] [0]是顶点的x坐标，v[] [1]是顶点的y坐标，v[] [2]是顶点的z坐标。

通过简单的减法从一个顶点的每个坐标到另一个顶点每个坐标我们得到了那个VECTOR。v1[0] = v[0][0] - v[1][0]，这计算现存的从一个顶点到另一个顶点的向量的X组件，v1[1] = v[0][1] - v[1][1]将计算Y组件，v1[2] = v[0][2] - v[1][2]计算Z组件等等。现在我们有了两个向量，所以我们计算它们的交叉乘积得到那个三角形的法线。

交叉相乘的规则是：

$$\text{out}[x] = v1[y] * v2[z] - v1[z] * v2[y]$$

$$\text{out}[y] = v1[z] * v2[x] - v1[x] * v2[z]$$

$$\text{out}[z] = v1[x] * v2[y] - v1[y] * v2[x]$$

我们最终得到了这个三角形的法线in out[]。

```
void calcNormal(float v[3][3], float out[3]) // 用三点计算一个立方体法线
{
    float v1[3], v2[3]; // 向量 1 (x,y,z) 和向量 2 (x,y,z)
    static const int x = 0; // 定义 X坐标
    static const int y = 1; // 定义 Y坐标
    static const int z = 2; // 定义 Z坐标
```

```
// 用减法在两点之间得到向量
// 从一点到另一点的X , Y , Z坐标
// 计算点1到点0的向量
```

```

v1[x] = v[0][x] - v[1][x];
v1[y] = v[0][y] - v[1][y];
v1[z] = v[0][z] - v[1][z];
// 计算点2到点1的向量
v2[x] = v[1][x] - v[2][x];
v2[y] = v[1][y] - v[2][y];
v2[z] = v[1][z] - v[2][z];
// 计算交叉乘积为我们提供一个表面的法线
out[x] = v1[y]*v2[z] - v1[z]*v2[y];
out[y] = v1[z]*v2[x] - v1[x]*v2[z];
out[z] = v1[x]*v2[y] - v1[y]*v2[x];

ReduceToUnit(out);           // 规格化向量
}

```

下面的例子正好用gluLookAt设立了一个观察点。我们设置一个观察点放置在0 , 5 , 50位置——正照看0 , 0 , 0并且所属的向上的向量正仰望 ( 0 , 1 , 0 ) ! : D

```

void ProcessHelix()           // 绘制一个螺旋
{
    GLfloat x;               // 螺旋x坐标
    GLfloat y;               // 螺旋y坐标
    GLfloat z;               // 螺旋z坐标
    GLfloat phi;              // 角
    GLfloat theta;             // 角
    GLfloat v,u;              // 角
    GLfloat r;                // 螺旋半径
    int twists = 5;            // 5个螺旋

    GLfloat glfMaterialColor[]={0.4f,0.2f,0.8f,1.0f};      // 设置材料色彩
    GLfloat specular[]={1.0f,1.0f,1.0f,1.0f};           // 设置镜象灯光

    glLoadIdentity();          // 重置Modelview矩阵
    gluLookAt(0, 5, 50, 0, 0, 0, 0, 1, 0);           // 场景 ( 0 , 0 , 0 ) 的视点中心 (0,5,50) , Y轴向上

    glPushMatrix();           // 保存Modelview矩阵

    glTranslatef(0,0,-50);        // 移入屏幕50个单位
    glRotatef(angle/2.0f,1,0,0);   // 在X轴上以1/2角度旋转
    glRotatef(angle/3.0f,0,1,0);   // 在Y轴上以1/3角度旋转
}

```

```
glMaterialfv(GL_FRONT_AND_BACK,GL_AMBIENT_AND_DIFFUSE,gfMaterialColor);
glMaterialfv(GL_FRONT_AND_BACK,GL_SPECULAR,specular);
```

然后我们计算螺旋的公式并给弹簧着色。十分简单，我就不再解释了，因为它不是这篇指南的主要目的。这段螺旋代码经过软件赞助者的许可被借用（并作了一点优化）。这是写作的简单的方法，但不是最快的方法。使用顶点数组可以使它更快！

```
r=1.5f; // 半径

glBegin(GL_QUADS); // 开始绘制立方体
for(phi=0; phi <= 360; phi+=20.0) // 以20度的间隔绘制
{
    for(theta=0; theta<=360*twists; theta+=20.0)
    {
        v=(phi/180.0f*3.142f); // 计算第一个点(0)的角度
        u=(theta/180.0f*3.142f); // 计算第一个点(0)的角度

        x=float(cos(u)*(2.0f+cos(v)))*r; // 计算x的位置(第一个点)
        y=float(sin(u)*(2.0f+cos(v)))*r; // 计算y的位置(第一个位置)
        z=float(((u-(2.0f*3.142f))+sin(v))*r); // 计算z的位置(第一个位置)

        vertexes[0][0]=x; // 设置第一个顶点的x值
        vertexes[0][1]=y; // 设置第一个顶点的y值
        vertexes[0][2]=z; // 设置第一个顶点的z值

        v=(phi/180.0f*3.142f); // 计算第二个点(0)的角度
        u=((theta+20)/180.0f*3.142f); // 计算第二个点(20)的角度

        x=float(cos(u)*(2.0f+cos(v)))*r; // 计算x位置(第二个点)
        y=float(sin(u)*(2.0f+cos(v)))*r; // 计算y位置(第二个点)
        z=float(((u-(2.0f*3.142f))+sin(v))*r); // 计算z位置(第二个点)

        vertexes[1][0]=x; // 设置第二个顶点的x值
        vertexes[1][1]=y; // 设置第二个顶点的y值
        vertexes[1][2]=z; // 设置第二个顶点的z值

        v=((phi+20)/180.0f*3.142f); // 计算第三个点(20)的角度
        u=((theta+20)/180.0f*3.142f); // 计算第三个点(20)的角度
```

```
x=float(cos(u)*(2.0f+cos(v)))*r; // 计算x位置(第三个点)
y=float(sin(u)*(2.0f+cos(v)))*r; // 计算y位置(第三个点)
z=float((( u-(2.0f*3.142f)) + sin(v) ) * r); // 计算z位置(第三个点)
```

```
vertexes[2][0]=x; // 设置第三个顶点的x值
vertexes[2][1]=y; // 设置第三个顶点的y值
vertexes[2][2]=z; // 设置第三个顶点的z值
```

```
v=((phi+20)/180.0f*3.142f); // 计算第四个点(20)的角度
u=((theta)/180.0f*3.142f); // 计算第四个点(0)的角度
```

```
x=float(cos(u)*(2.0f+cos(v)))*r; // 计算x位置(第四个点)
y=float(sin(u)*(2.0f+cos(v)))*r; // 计算y位置(第四个点)
z=float((( u-(2.0f*3.142f)) + sin(v) ) * r); // 计算z位置(第四个点))
```

```
vertexes[3][0]=x; // 设置第四个顶点的x值
vertexes[3][1]=y; // 设置第四个顶点的y值
vertexes[3][2]=z; // 设置第四个顶点的z值
```

```
calcNormal(vertexes,normal); // 计算立方体的法线
```

```
glNormal3f(normal[0],normal[1],normal[2]); // 设置法线
```

```
// 渲染四边形
```

```
glVertex3f(vertexes[0][0],vertexes[0][1],vertexes[0][2]);
glVertex3f(vertexes[1][0],vertexes[1][1],vertexes[1][2]);
glVertex3f(vertexes[2][0],vertexes[2][1],vertexes[2][2]);
glVertex3f(vertexes[3][0],vertexes[3][1],vertexes[3][2]);
```

```
}
```

```
}
```

```
glEnd(); // 绘制结束
```

```
glPopMatrix(); // 取出矩阵
```

```
}
```

这两个事例 ( ViewOrtho and ViewPerspective ) 被编码以使它变得很容易地在一个直交的情形下绘制并且不费力的返回透视图。

ViewOrtho简单地设立了这个射影矩阵，然后增加一份现行射影矩阵的拷贝到OpenGL栈上。这个恒等矩阵然后被装载并且当前屏幕正投影观察决议被提出。

利用2维坐标以屏幕左上角0 , 0和屏幕右下角639 , 479来绘制是可能的。

最后，modelview矩阵为透视材料激活。

ViewPerspective设置射影矩阵模式取回ViewOrtho在堆栈上推进的非正交矩阵。然后样本视图被选择因此我们可以透视材料。

我建议你保留这两个过程，能够着色2D而不需担心射影矩阵很不错。

```
void ViewOrtho()                                // 设置一个z正视图
{
    glMatrixMode(GL_PROJECTION);                // 选择投影矩阵
    glPushMatrix();                            // 保存当前矩阵
    glLoadIdentity();                          // 重置矩阵
    glOrtho( 0, 640 , 480 , 0, -1, 1 );      // 选择标准模式
    glMatrixMode(GL_MODELVIEW);                // 选择样本视图矩阵
    glPushMatrix();                            // 保存当前矩阵
    glLoadIdentity();                          // 重置矩阵
}

void ViewPerspective()                           // 设置透视视图
{
    glMatrixMode( GL_PROJECTION );            // 选择投影矩阵
    glPopMatrix();                            // 取出矩阵
    glMatrixMode( GL_MODELVIEW );             // 选择模型变换矩阵
    glPopMatrix();                            // 弹出矩阵
}
```

现在是解释那个冒牌的辐射状的模糊效果是如何作的时候了。

我们需要绘制这个场景——它从中心开始在所有方向上模糊出现。窍门是在没有主要的性能瓶颈的情况下做出的。我们不能读写象素，并且如果我们想和非kick-but视频卡兼容，我们不能使用扩展名何驱动程序特殊命令。

没办法了吗？

不，解决方法非常简单，OpenGL赋予我们“模糊”纹理的能力。OK……并非真正的模糊，但我们利用线性过滤去依比例决定一个纹理，结果（有些想象成分）看起来象高斯模糊。

因此如果我们正确地在3D场景中放了大量的被拉伸的纹理并依比例决定会有什么发生？答案比你想象的还简单。

问题一：透视一个纹理

有一个后缓冲器在象素格式下问题容易解决。在没有后缓冲器的情况下透视一个纹理在眼睛看来是一个真正的痛苦。

透视纹理刚好借助一个函数来完成。我们需要绘制我们的实体然后利用glCopytexImage函数复制这个结果（在交换前，后缓冲器之前）后到纹理。

问题二：在3D实体前精确地协调纹理。

我们知道：如果我们在没有设置正确的透视的情况下改变了视口，我们就得到一个我们的实体的一个被拉伸的透视图。例如如果我们设置一个是视口足够宽我们就得到一个垂直地被拉伸的透视图。

解决方法是首先设置一个视口正如我们的纹理（ $128 \times 128$ ）。透视我们的实体到这个纹理之后，我们利用当前屏幕决议着色这个纹理到屏幕。这种方法OpenGL缩减这个实体去适应纹理，并且我们拉伸纹理到全屏大小时，OpenGL重新调整纹理的大小去完美的适应在我们的3d实体顶端。希望我没有丢掉任何一点。另一个灵活的例子是，如果你取一个 $640 \times 480$ 大小screenshot，然后调整成为 $256 \times 256$ 的位图，你可以以一个纹理装载这个位图，并拉伸它使之适合 $640 \times 480$ 的屏幕。这个质量可能不会以前一样好，但是这个纹理排列起的效果应当接近最初的 $640 \times 480$ 图象。

On to the fun stuff! 这个函数相当简单，并且是我的首选的“设计窍门”之一。它设置一个与我们的BlurTexture度数相匹配的大小的视口。然后它被弹簧的着色程序调用。弹簧将由于视口被拉伸适应 $128 \times 128$ 的纹理。

在弹簧被拉伸至 $128 \times 128$ 视口大小之后，我们约定BlurTexture且用glCopyTexImage2D从视口拷贝色彩缓冲器到BlurTexture。

参数如下：

GL\_TEXTURE\_2D指出我们正使用一个2Dimensional纹理，0是我们想要拷贝缓冲器到mip的绘图等级，默认等级是0。GL\_LUMINANCE指出被拷贝的数据格式。我之所以使用GL\_LUMINANCE因为最终结果看起来比较好。这种情形缓冲器的亮度部分将被拷贝到纹理。其它参数可以是GL\_ALPHA, GL\_RGB, GL\_INTENSITY等等。

其次的两个参数告诉OpenGL从(0,0)开始拷贝到哪里。宽度和高度(128,128)是从左到右有多少象素要拷贝并且上下拷贝多少。最后一个参数仅用来指出我们是否想要一个边界——哪个不想要。

既然在我们的BlurTexture我们已经有了一个色彩缓冲器的副本（和被拉伸的弹簧一致），我们可以清除那个缓冲器，向后设置那个视口到适当的度数（ $640 \times 480$ 全屏）。

重要：

这个窍门能用在只有双缓冲器象素格式的情况下。原因是所有这些操作从观察者面前被

隐藏起来。（在后缓冲器完成）。

```

void RenderToTexture() // 着色到一个纹理
{
    glViewport(0,0,128,128); // 设置我们的视口

    ProcessHelix(); // 着色螺旋

    glBindTexture(GL_TEXTURE_2D,BlurTexture); // 绑定模糊纹理

    // 拷贝我们的视口到模糊纹理 (从 0,0 到 128,128... 无边界)
    glCopyTexImage2D(GL_TEXTURE_2D, 0, GL_LUMINANCE, 0, 0, 128, 128, 0);

    glClearColor(0.0f, 0.0f, 0.5f, 0.5); // 调整清晰的色彩到中等蓝色
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清屏和深度缓冲

    glViewport(0,0,640,480); // 调整视口 (0,0 to 640x480)
}

```

DrawBlur函数仅在我们的3D场景前绘制一些混合的方块——用BlurTexture我们以前已实现。这样，借由阿尔发和缩放这个纹理，我们得到了真正看起来象辐射状的模糊的效果。

我首先禁用GEN\_S 和 GEN\_T（我沉溺于球体影射，因此我的程序通常启用这些指令：P）。

我们启用2D纹理，禁用深度测试，调整正确的函数，起用混合然后约束BlurTexture。下一件我们要作的事情是转换到标准视图，那样比较容易绘制一些完美适应屏幕大小的方块。这是我们在3D实体顶端排列纹理的方法（通过拉伸纹理匹配屏幕比例）。这是问题二要解决的地方。

```

void DrawBlur(int times, float inc) // 绘制模糊的图象
{
    float spost = 0.0f; // 纹理坐标偏移量
    float alphainc = 0.9f / times; // alpha混合的衰减量
    float alpha = 0.2f; // Alpha初值

    // 禁用自动生成纹理坐标
    glDisable(GL_TEXTURE_GEN_S);
    glDisable(GL_TEXTURE_GEN_T);

```

```

glEnable(GL_TEXTURE_2D);           // 启用 2D 纹理映射
glDisable(GL_DEPTH_TEST);         // 深度测试不可用
glBlendFunc(GL_SRC_ALPHA,GL_ONE); // 设置混合模式
glEnable(GL_BLEND);              // 启用混合
glBindTexture(GL_TEXTURE_2D,BlurTexture); // 绑定混合纹理
ViewOrtho();                     // 切换到标准视图

alphainc = alpha / times;        // 减少alpha值

```

我们多次绘制这个纹理用于创建那个辐射效果，缩放这个纹理坐标并且每次我们做另一个关口时增大混合因数。我们绘制25个方块，每次按照0.015f拉伸这个纹理。

```

glBegin(GL_QUADS);               // 开始绘制方块
for (int num = 0;num < times;num++)
{
    glColor4f(1.0f, 1.0f, 1.0f, alpha); // 调整alpha值
    glTexCoord2f(0+spost,1-spost);
    glVertex2f(0,0);

    glTexCoord2f(0+spost,0+spost);
    glVertex2f(0,480);

    glTexCoord2f(1-spost,0+spost);
    glVertex2f(640,480);

    glTexCoord2f(1-spost,1-spost);
    glVertex2f(640,0);

    spost += inc;                  // 逐渐增加 spost (快速靠近纹理中心)
    alpha = alpha - alphainc;      // 逐渐增加 alpha (逐渐淡出纹理)
}
glEnd();                         // 完成绘制方块

ViewPerspective();                // 转换到一个透视视图

glEnable(GL_DEPTH_TEST);          // 深度测试可用
glDisable(GL_TEXTURE_2D);         // 2D纹理映射不可用
glDisable(GL_BLEND);              // 混合不可用
glBindTexture(GL_TEXTURE_2D,0);    // 释放模糊纹理

```

}

瞧，这是以前从未见过的最短的绘制程序，有很棒的视觉效果！

我们调用RenderToTexture 函数。幸亏我们视口改变这个函数才着色被拉伸的弹簧。对于我们的纹理拉伸的弹簧被着色，并且这些缓冲器被清除。

我们之后绘制“真正的”弹簧(你在屏幕上看到的3D实体)通过调用 ProcessHelix( )。

最后我们在弹簧前面绘制一些混合的方块。有织纹的方块将被拉伸以适应在真正的3D弹簧上面。

```
void Draw (void)                                // 绘制场景
{
    glClearColor(0.0f, 0.0f, 0.0f, 0.5);           // 将清晰的颜色设定为黑色
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除屏幕和深度缓冲器
    glLoadIdentity();                            // 重置视图
    RenderToTexture();                         // 着色纹理
    ProcessHelix();                           // 绘制我们的螺旋
    DrawBlur(25,0.02f);                      // 绘制模糊效果
    glFlush ();                               // 强制OpenGL绘制我们所有的图形
}
```

我希望你满意这篇指南，它实在没有比透视一个纹理讲授更多其它内容，但它是一个干脆地添加到你的3D应用程序中有趣的效果。

如果你有任何的注释建议或者如果你知道一种更好的方法执行这个效果联系我  
rio@spinningkids.org。

我也想要委托你去做一列事情(家庭作业)：D

1) 更改DrawBlur程序变为一个水平的模糊之物，垂直的模糊之物和一些更好的效果。  
(转动模糊之物！)。

2) 玩转DrawBlur参数(添加，删除)变为一个好的程序和你的音乐同步。

3) 用GL\_LUMINANCE玩弄DrawBlur参数和一个SMALL纹理(惊人的光亮！)。

4) 用暗色纹理代替亮色尝试大骗(哈哈，自己造的)测定体积的阴影。

好了，这应该是所有的了(到此为止)。

访问我的站点<http://www.spinningkids.org/rio>.

获得更多的最新指南。



## 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

## 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

## 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

## 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

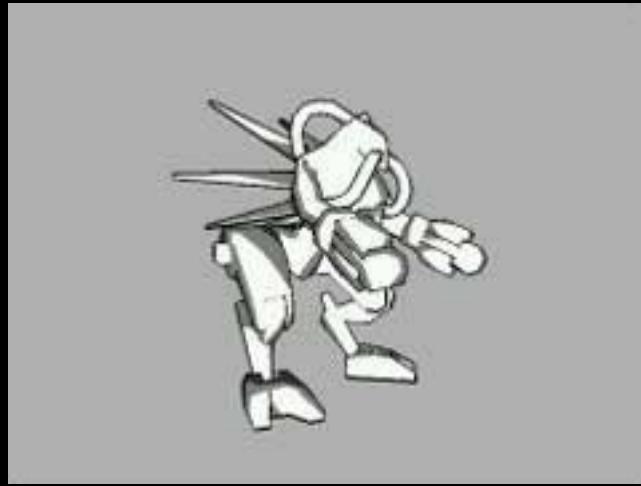
## 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。



## 第37课



卡通映射:

什么是卡通了，一个轮廓加上少量的几种颜色。使用一维纹理映射，你也可以实现这种效果。

看到人们仍然e-mail我请求在文章中使用我方才在GameDev.net上写的源代码，还看到文章的第二版（在那每一个API附带源码）不是在中途完成之前连贯的结束。我已经把这篇指南一并出租给了NeHe（这实际上是写文章的最初意图）因此你们所有的OpenGL领袖可以玩转它。对模型的选择表示抱歉，但是我最近一直在玩Quake 2。

注释：这篇文章的源代码可以在这里找到：

<http://www.gamedev.net/reference/programming/features/celshading>.

这篇指南实际上并不解释原理，仅仅解释代码。在上面的连接中可以发现为什么它能工作。现在不断地大声抱怨STOP E-MAILING ME REQUESTS FOR SOURCE CODE!!!!

首先，我们需要包含一些额外的头文件。第一个（math.h）我们可以使用sqrtf (square root)函数，第二个用来访问文件。

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

现在我们将定义一些结构体来帮助我们存贮我们的数据（保存好几百浮点数组）。第一个是tagMATRIX结构体。如果你仔细地看，你将看到我们正象包含一个十六个浮点数的1维数组~一个2维 $4 \times 4$ 数族一样存储那个矩阵。这下至OpenGL存储它的矩阵的方式。如果我们使用 $4 \times 4$ 数组，这些值将发生错误的顺序。

```
typedef struct tagMATRIX           // 保存OpenGL矩阵的结构体
{
    float Data[16];              // 由于OpenGL的矩阵的格式我们使用[16
}                                MATRIX;
```

第二是向量的类。仅存储X, Y和Z的值

```
typedef struct tagVECTOR           // 存储一个单精度向量的结构体
{
    float X, Y, Z;               // 向量的分量
}                                VECTOR;
```

第三，我们持有顶点的结构。每一个顶点仅需要它的法线和位置（没有纹理的现行纵坐标）信息。它们必须以这样的次序被存放，否则当它停止装载文件的事件将发生严重的错误（我发现艰难的情形：（教我分块出租我的代码。）。

```
typedef struct tagVERTEX           // 存放单一顶点的结构
{
    VECTOR Nor;                 // 顶点法线
    VECTOR Pos;                  // 顶点位置
}                                VERTEX;
```

最后是多边形的结构。我知道这是存储顶点的愚蠢的方法，要不是它完美工作的简单的缘故。通常我愿意使用一个顶点数组，一个多边形数组，和包括一个在多边形中的3个顶点的指数，但这比较容易显示你想干什么。

```
typedef struct tagPOLYGON           // 存储单一多边形的结构
{
    VERTEX Verts[3];               // 3个顶点结构数组
} POLYGON;
```

优美简单的材料也在这里了。为每一个变量的一个解释考虑那个注释。

```
bool      outlineDraw   = true;          // 绘制轮廓的标记
bool      outlineSmooth = false;         // Anti-Alias 线段的标记
float     outlineColor[3] = { 0.0f, 0.0f, 0.0f }; // 线段的颜色
float     outlineWidth   = 3.0f;          // 线段的宽度

VECTOR    lightAngle;                  // 灯光的方向
bool      lightRotate  = false;         // 是否我们旋转灯光的标记

float     modelAngle   = 0.0f;          // 模型的Y轴角度
bool      modelRotate  = false;         // 旋转模型的标记

POLYGON   *polyData   = NULL;           // 多边形数据
int       polyNum    = 0;                // 多边形的编号

GLuint    shaderTexture[1];             // 存储纹理ID
```

这是得到的再简单不过的模型文件格式。最初的少量字节存储在场景中的多边形的编号，文件的其余是tagPOLYGON结构体的一个数组。正因如此，数据在没有任何需要去分类到详细的顺序的情况下被读出。

```
BOOL ReadMesh ()                      // 读“model.txt”文件
```

```

{
    FILE *In = fopen ("Data\\model.txt", "rb");      // 打开文件

    if (!In)
        return FALSE;           // 如果文件没有打开返回 FALSE

    fread (&polyNum, sizeof (int), 1, In);    // 读文件头，多边形的个数

    polyData = new POLYGON [polyNum];          // 分配内存

    fread (&polyData[0], sizeof (POLYGON) * polyNum, 1, In); // 把所有多边形的数据读入

    fclose (In);                // 关闭文件

    return TRUE;                // 工作完成
}

```

一些基本的数学函数而已。DotProduct计算2个向量或平面之间的角，Magnitude函数计算向量的长度，Normalize函数缩放向量到一个单位长度。

```

inline float DotProduct (VECTOR &V1, VECTOR &V2)      //计算两个向量之间的角度
{
    return V1.X * V2.X + V1.Y * V2.Y + V1.Z * V2.Z;
}

inline float Magnitude (VECTOR &V)                  // 计算向量的长度
{
    return sqrtf (V.X * V.X + V.Y * V.Y + V.Z * V.Z);
}

void Normalize (VECTOR &V)                          // 创建一个单位长度的向量
{
    float M = Magnitude (V);

    if (M != 0.0f)           // 确保我们没有被0隔开
    {
        V.X /= M;
        V.Y /= M;
        V.Z /= M;
    }
}

```

}

这个函数利用给定的矩阵旋转一个向量。请注意它仅旋转这个向量——与向量的位置相比它算不了什么。它用来当旋转法线确保当我们在计算灯光时它们停留在正确的方向上。

```
void RotateVector (MATRIX &M, VECTOR &V, VECTOR &D)      // 利用提供的矩阵旋转一个向量
{
    D.X = (M.Data[0] * V.X) + (M.Data[4] * V.Y) + (M.Data[8] * V.Z);
    D.Y = (M.Data[1] * V.X) + (M.Data[5] * V.Y) + (M.Data[9] * V.Z);
    D.Z = (M.Data[2] * V.X) + (M.Data[6] * V.Y) + (M.Data[10] * V.Z);
}
```

引擎的第一个主要的函数…… 初始化，按所说的精确地做。我已经砍掉了在注释中不再需要的代码段。

```
// 一些GL 初始代码和用户初始化从这里开始
BOOL Initialize (GL_Window* window, Keys* keys)
{
```

这3个变量用来装载着色文件。在文本文件中为了单一的线段线段包含了空间，虽然 shaderData存储了真实的着色值。你可能奇怪为什么我们的96个值被32个代替了。好了，我们需要转换greyscale 值为RGB以便OpenGL能使用它们。我们仍然可以以greyscale存储这些值，但向上负载纹理时我们至于R，G和B成分仅仅使用同一值。

```
char Line[255];          // 255个字符的存储量
float shaderData[32][3]; // 96个着色值的存储量
g_window    = window;
g_keys      = keys;
FILE *In = NULL;         // 文件指针
```

当绘制线条时，我们想要确保很平滑。初值被关闭，但是按“2”键，它可以被toggled on/off。

```
glShadeModel (GL_SMOOTH);           // 使用色彩阴影平滑  
glDisable (GL_LINE_SMOOTH);         // 线条平滑初始化不可用  
  
glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // 提高计算精度  
glClearColor (0.7f, 0.7f, 0.7f, 0.0f);    // 设置为灰色背景  
glClearDepth (1.0f);                  // 设置深度缓存值  
glEnable (GL_DEPTH_TEST);            // 启用深度测试  
glDepthFunc (GL_LESS);              // 设置深度比较函数  
glShadeModel (GL_SMOOTH);           // 启用反走样  
glDisable (GL_LINE_SMOOTH);  
  
glEnable (GL_CULL_FACE);             // 启用剔除多边形功能
```

我们使 OpenGL灯光不可用因为我们自己做所以的灯光计算。

```
glDisable (GL_LIGHTING);           // 使 OpenGL 灯光不可用
```

这里是我们装载阴影文件的地方。它简单地以32个浮点值ASCII码存放（为了轻松修改），每一个在separate线上。

```
In = fopen ("Data\\shader.txt", "r");      // 打开阴影文件  
  
if (In)                                // 检查文件是否打开  
{  
    for (i = 0; i < 32; i++)               // 循环32次  
    {  
        if (feof (In))                  // 检查文件是否结束  
            break;
```

```
fgets (Line, 255, In); // 获得当前线条
```

这里我们转换 greyscale 值为 RGB, 正象上面所描述的。

```
// 从头到尾复制这个值
shaderData[i][0] = shaderData[i][1] = shaderData[i][2] = atof (Line);
}

fclose (In); // 关闭文件
}

else
return FALSE;
```

现在我们向上装载这个纹理。同样它清楚地规定，不要使用任何一种过滤在纹理上否则它看起来奇怪，至少可以这样说。GL\_TEXTURE\_1D被使用因为它是值的一维数组。

```
glGenTextures (1, &shaderTexture[0]); // 获得一个自由的纹理ID

glBindTexture (GL_TEXTURE_1D, shaderTexture[0]); // 绑定这个纹理。从现在开始它变为一维

// 使用邻近点过滤
glTexParameteri (GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
glTexParameteri (GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);

// 设置纹理
glTexImage1D (GL_TEXTURE_1D, 0, GL_RGB, 32, 0, GL_RGB, GL_FLOAT, shaderData);
```

现在调整灯光方向。我已经使得它向下指向Z轴正方向，这意味着它将正面碰撞模型

```
lightAngle.X = 0.0f;
lightAngle.Y = 0.0f;
lightAngle.Z = 1.0f;

Normalize (lightAngle);
```

读取Mesh文件,并返回

```
} return ReadMesh (); // 读取Mesh文件,并返回
```

与上面的函数相对应……卸载，删除由Initialize 和 ReadMesh 创建的纹理和多边形数据。

```
void Deinitialize (void)
{
    glDeleteTextures (1, &shaderTexture[0]); // 删除阴影纹理

    delete [] polyData; // 删除多边形数据
}
```

主要的演示循环。所有这些用来处理输入和更新角度。控制如下：

<SPACE> = 锁定旋转

1 = 锁定轮廓绘制

2 = 锁定轮廓 anti-aliasing

<UP> = 增加线宽

<DOWN> = 减小线宽

```
void Update (DWORD milliseconds) // 这里执行动作更新
```

```
{  
    if (g_keys->keyDown [' '] == TRUE)          // 空格是否被按下  
    {  
        modelRotate = !modelRotate;           // 锁定模型旋转开/关  
  
        g_keys->keyDown [' '] = FALSE;  
    }  
  
    if (g_keys->keyDown ['1'] == TRUE)          // 1是否被按下  
    {  
        outlineDraw = !outlineDraw;           // 切换是否绘制轮廓线  
  
        g_keys->keyDown ['1'] = FALSE;  
    }  
  
    if (g_keys->keyDown ['2'] == TRUE)          // 2是否被按下  
    {  
        outlineSmooth = !outlineSmooth;       // 切换是否使用反走样  
  
        g_keys->keyDown ['2'] = FALSE;  
    }  
  
    if (g_keys->keyDown [VK_UP] == TRUE)         // 上键增加线的宽度  
    {  
        outlineWidth++;  
  
        g_keys->keyDown [VK_UP] = FALSE;  
    }  
  
    if (g_keys->keyDown [VK_DOWN] == TRUE)         // 下减少线的宽度  
    {  
        outlineWidth--;  
  
        g_keys->keyDown [VK_DOWN] = FALSE;  
    }  
  
    if (modelRotate)                            // 是否旋转  
        modelAngle += (float) (milliseconds) / 10.0f; // 更新旋转角度  
}
```

你一直在等待的函数。Draw 函数做每一件事情——计算阴影的值，着色网孔，着色轮廓，等等，这是它作的。

```
void Draw (void)
{
```

TmpShade用来存储当前顶点的色度值。所有顶点数据同时被计算，意味着我们只需使用我们能继续使用的单个的变量。

TmpMatrix, TmpVector 和 TmpNormal同样被用来计算顶点数据，TmpMatrix在函数开始时被调整一次并一直保持到Draw函数被再次调用。TmpVector 和 TmpNormal则相反，当另一个顶点被处理时改变。

```
float TmpShade; // 临时色度值
MATRIX TmpMatrix; // 临时 MATRIX 结构体
VECTOR TmpVector, TmpNormal; // 临时 VECTOR结构体
```

清除缓冲区矩阵数据

```
glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // 清除缓冲区
glLoadIdentity (); // 重置矩阵
```

首先检查我们是否想拥有平滑的轮廓。如果是，我们就打开anti-aliasing。否则把它关闭。简单！

```
if (outlineSmooth) // 检查我们是否想要 Anti-Aliased 线条
{
    glHint (GL_LINE_SMOOTH_HINT, GL_NICEST); // 启用它们
    glEnable (GL_LINE_SMOOTH);
}
```

```
else // 否则不启用
    glDisable(GL_LINE_SMOOTH);
```

然后我们设置视口。我们反向移动摄像机2个单元，之后以一定角度旋转模型。注：由于我们首先移动摄像机，这个模型将在现场旋转。如果我们以另一种方法做，模型将绕摄像机旋转。

我们之后从OpenGL中取最新创建的矩阵并把它存储在 TmpMatrix。

```
glTranslatef(0.0f, 0.0f, -2.0f); // 移入屏幕两个单位
glRotatef(modelAngle, 0.0f, 1.0f, 0.0f); // 绕Y轴旋转这个模型

glGetFloatv(GL_MODELVIEW_MATRIX, TmpMatrix.Data); // 获得产生的矩阵
```

戏法开始了。首先我们启用一维纹理，然后启用着色纹理。这被OpenGL用来当作一个 look-up表格。我们之后调整模型的颜色（白色）我选择白色是因为它亮度高并且描影法比其它颜色好。我建议你不要使用黑色：）

```
// 卡通渲染代码
 glEnable(GL_TEXTURE_1D); // 启用一维纹理
 glBindTexture(GL_TEXTURE_1D, shaderTexture[0]); // 锁定我们的纹理

 glColor3f(1.0f, 1.0f, 1.0f); // 调整模型的颜色
```

现在我们开始绘制那些三角形。尽管我们看到在数组中的每一个多边形，然后旋转它的每一个顶点。第一步是拷贝法线信息到一个临时的结构中。因此我们能旋转法线，但仍然保留原来保存的值（没有精确降级）。

```
glBegin(GL_TRIANGLES); // 告诉OpenGL 我们即将绘制三角形

for (i = 0; i < polyNum; i++)
{

```

```

for (j = 0; j < 3; j++)           // 从头到尾循环每一个顶点
{
    TmpNormal.X = polyData[i].Verts[j].Nor.X; // 用当前顶点的法线值填充
TmpNormal结构
    TmpNormal.Y = polyData[i].Verts[j].Nor.Y;
    TmpNormal.Z = polyData[i].Verts[j].Nor.Z;

```

第二，我们通过初期从OpenGL中攫取的矩阵来旋转这个法线。我们之后规格化因此它并不全部变为螺旋形。

```

// 通过矩阵旋转
RotateVector (TmpMatrix, TmpNormal, TmpVector);

Normalize (TmpVector); // 规格化这个新法线

```

第三，我们获得那个旋转的法线的点积灯光方向（称为lightAngle，因为我忘了从我的旧的light类中改变它）。我们之后约束这个值在0——1的范围。（从-1到+1）

```

// 计算色度值
TmpShade = DotProduct (TmpVector, lightAngle);

if (TmpShade < 0.0f)
    TmpShade = 0.0f; // 如果负值约束这个值到0

```

第四，对于OpenGL我们象忽略纹理坐标一样忽略这个值。阴影纹理与一个查找表一样来表现（色度值正成为指数），这是（我认为）为什么1D纹理被创造主要原因。对于OpenGL我们之后忽略这个顶点位置，并不断重复，重复。至此我认为你已经抓到了概念。

```

glTexCoord1f (TmpShade); // 规定纹理的纵坐标当作这个色度值
// 送顶点

```

```

        glVertex3fv (&polyData[i].Verts[j].Pos.X);
    }

}

glEnd (); // 告诉OpenGL 完成绘制

glDisable (GL_TEXTURE_1D); // 1D 纹理不可用

```

现在我们转移到轮廓之上。一个轮廓能以“它的相邻的边，一边为可见，另一边为不可见”定义。在OpenGL中，这是深度测试被规定小于或等于(GL\_LESS)当前值的地方，并且就在那时所有前面的面被精选。我们同样也要混合线条，以使它看起来不错：）

那么，我们使混合可用并规定混合模式。我们告诉OpenGL与着色线条一样着色backfacing多边形，并且规定这些线条的宽度。我们精选所有前面多边形，并规定测试深度小于或等于当前的Z值。在这个线条的颜色被规定后，我们从头到尾循环每一个多边形，绘制它的顶点。我们仅需忽略顶点位置，而不是法线或着色值因为我们需要的仅仅是轮廓。

```

// 轮廓代码
if (outlineDraw) // 检查看是否我们需要绘制轮廓
{
    glEnable (GL_BLEND); // 使混合可用
    // 调整混合模式
    glBlendFunc (GL_SRC_ALPHA ,GL_ONE_MINUS_SRC_ALPHA);

    glPolygonMode (GL_BACK, GL_LINE); // 绘制轮廓线
    glLineWidth (outlineWidth); // 调整线宽

    glCullFace (GL_FRONT); // 剔出前面的多边形

    glDepthFunc (GL_LESS); // 改变深度模式

    glColor3fv (&outlineColor[0]); // 规定轮廓颜色

    glBegin (GL_TRIANGLES); // 告诉OpenGL我们想要绘制什么

    for (i = 0; i < polyNum; i++) // 从头到尾循环每一个多边形
    {
        for (j = 0; j < 3; j++) // 从头到尾循环每一个顶点
        {

```

```
// 送顶点  
glVertex3fv (&polyData[i].Verts[j].Pos.X);  
}  
}  
  
glEnd(); // 告诉 OpenGL 我们已经完成
```

这样以后，我们就把它规定为以前的状态，然后退出

```
glDepthFunc(GL_LESS); // 重置深度测试模式  
glCullFace(GL_BACK); // 重置剔出背面多边形  
glPolygonMode(GL_BACK, GL_FILL); // 重置背面多边形绘制方式  
glDisable(GL_BLEND); // 混合不可用  
}  
}
```

你现在看到Cel-Shading并非那样难。当然技术可以提高非常多。一个好的例子是游戏 XIII [http://www.nvidia.com/object/game\\_xiii.html](http://www.nvidia.com/object/game_xiii.html)，它使你认为你在一个卡通世界里。如果你想在卡通透视技术里达到更深层次，你可以浏览这本书实时透视这一章“Non-Photorealistic Rendering”。如果你更喜欢在WEB上读论文，在这里可以发现一大堆链接列表：<http://www.red3d.com/cwr/npr/>

版权与使用声明：  
我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是 zhouwei02@mails.tsinghua.edu.cn , 如果你有任何问题,都可以联系我。

引子  
网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风



一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第36课

第38课 &gt;



## 第38课



从资源文件中载入图像:

如何把图像数据保存到\*.exe程序中，使用Windows的资源文件吧，它既简单又实用。

欢迎来到NeHe教程第38课。离上节课的写作已经有些时日了，加上写了一整天的code，也许笔头已经开始生锈了：)

现在你已经学会了如何做方格贴图，如何读入bitmap及各种光栅图像...那么如何做三角形贴图，又如何在.exe文件中体现你的纹理呢？

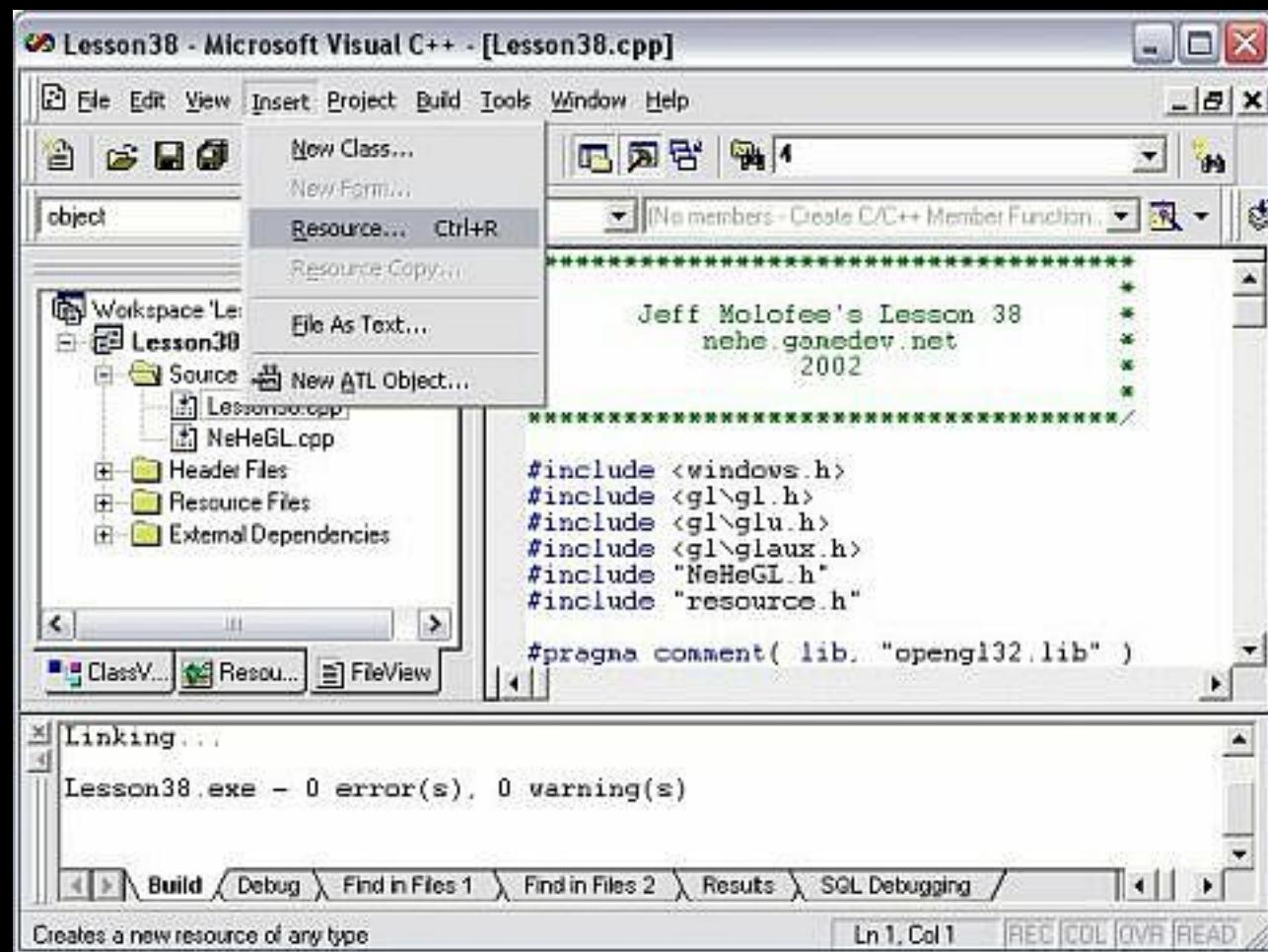
我每每被问及这两个问题，可是一旦你看到他们是多么简单，你就会大骂自己居然没有想到过：)

我不会事无巨细地解释每一个细节，只需给你一些抓图，就明白了。我将基于最新的code，请在主页"NeHeGL | Basecode"下或者这张网页最下面下载。

首先，我们把图像加载入资源文件。我向大家已经知道怎么做了，只是，你忽略了几步，于是值得到一些无用的资源文件。里面有bitmap文件，却无法使用。

还记得吧？我们使用Visual C++ 6.0 做的。如果你使用其它工具，这页教材关于资源的部分（尤其是那些图）完全不适用。

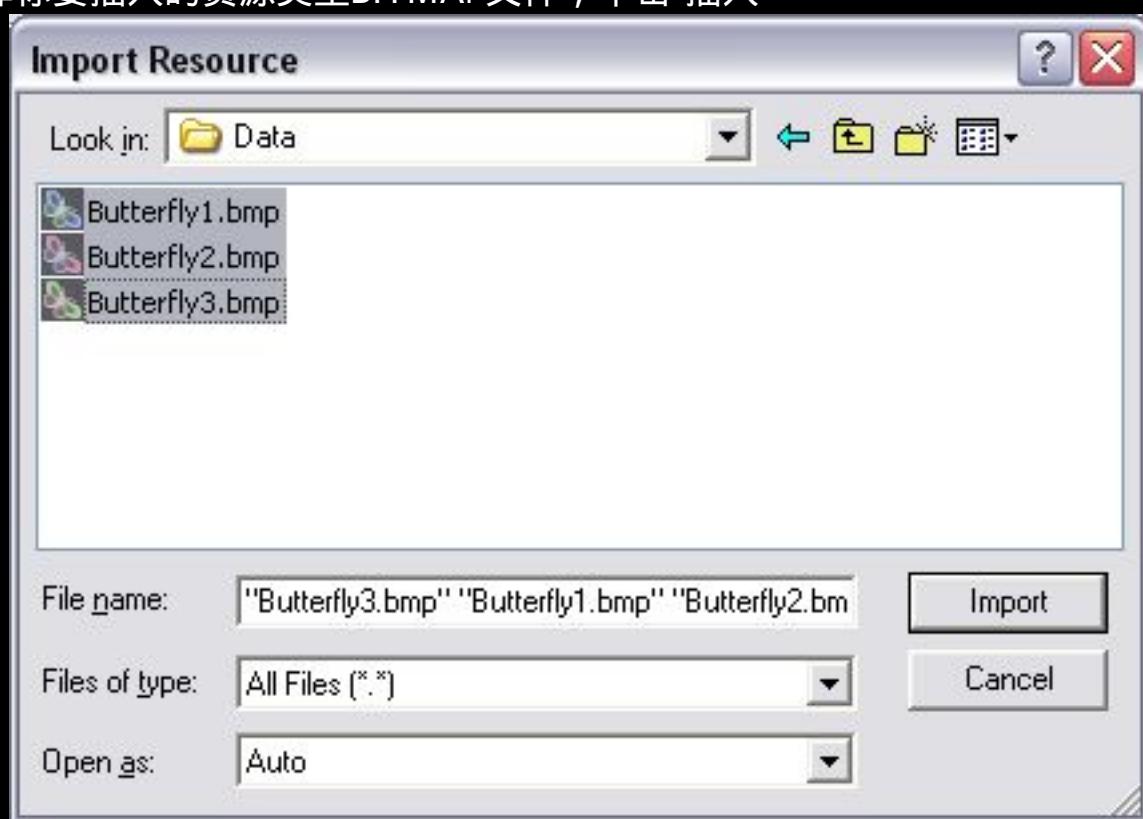
\* 暂时你只能用24bit BMP 图像。如果读8bit BMP文件要写很多额外的code。我很希望听到你们谁有更小的/更好的loader。我这里的读入8bit 和 24bit BMP 的code实在臃肿。用LoadImage就可以了。



打开文件，点击“插入”菜单，选“资源”



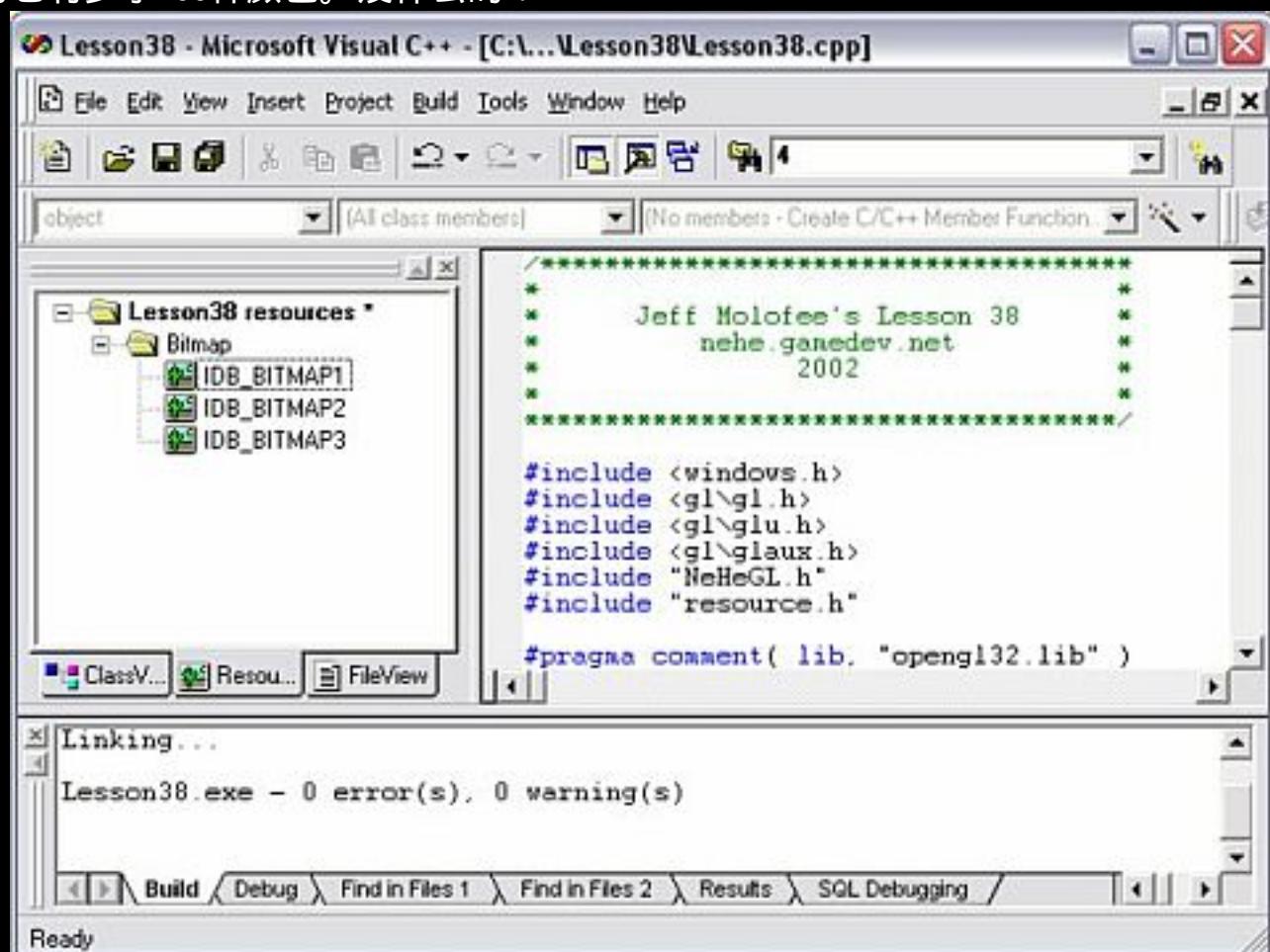
然后选择你要插入的资源类型BITMAP文件，单击“插入”



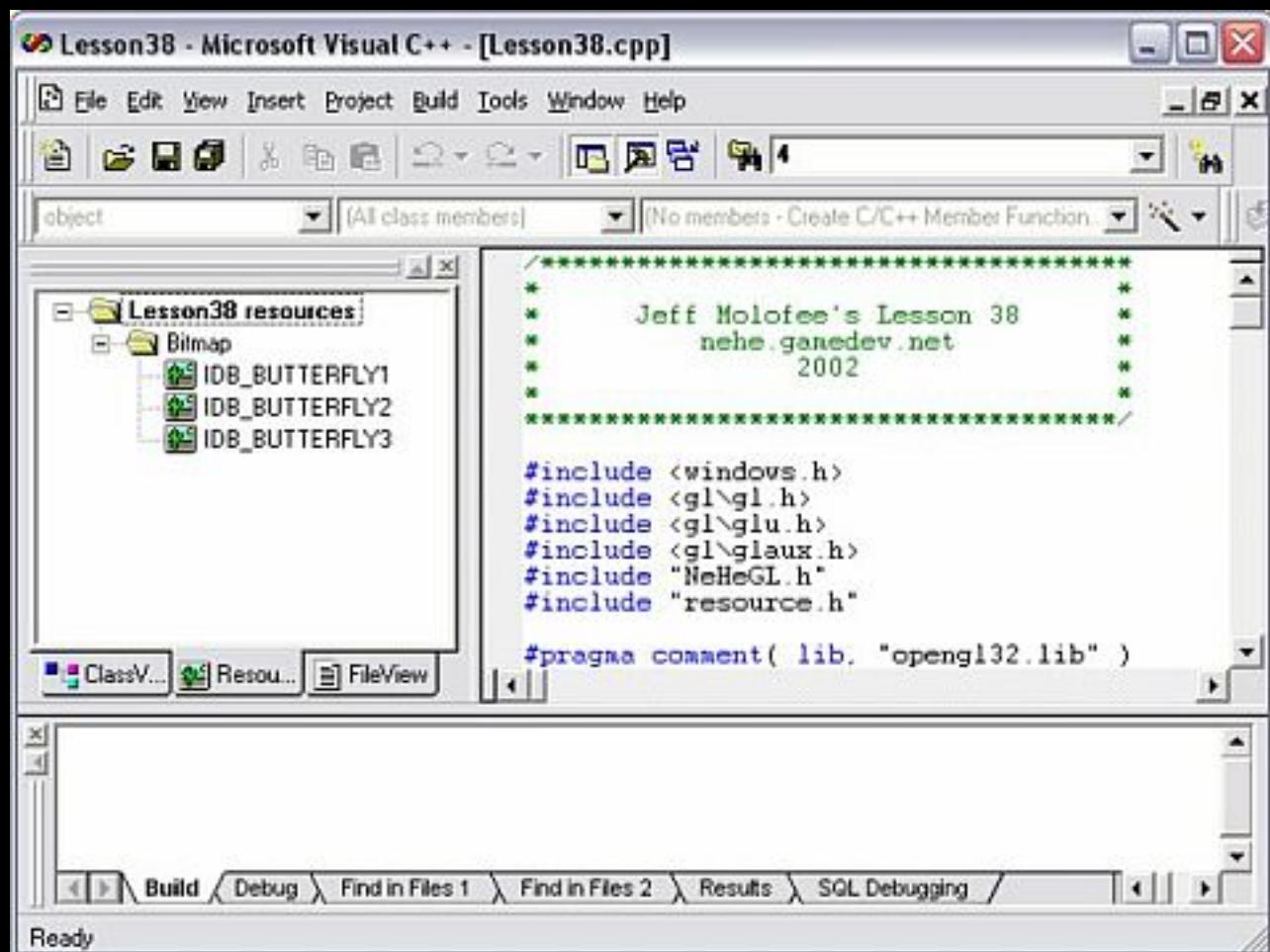
然后是文件窗口，进入DATA目录，选中三个图形文件（用Ctrl啦）然后点“读入”。注意文件类型是否正确。



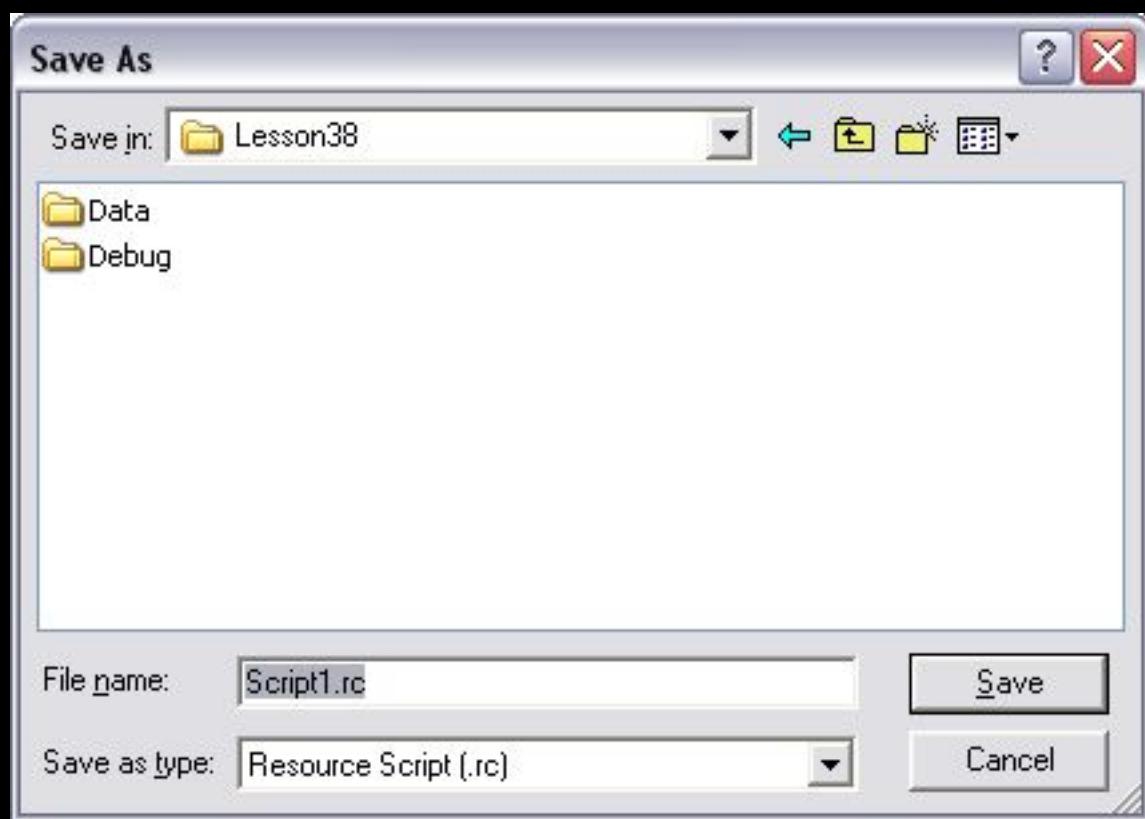
接下来会弹出三次警告（一个文件一次），说读入正确，但该文件不能被浏览或编辑，因为它有多于256种颜色。没什么的！



一旦所有图形都调入，将会出现一个列表。每个图分配有一个ID，每个ID都是IDB\_BITMAP打头的，然后数字1-3。你要是懒得改，就不用管它了。不过我们还都比较勤快！

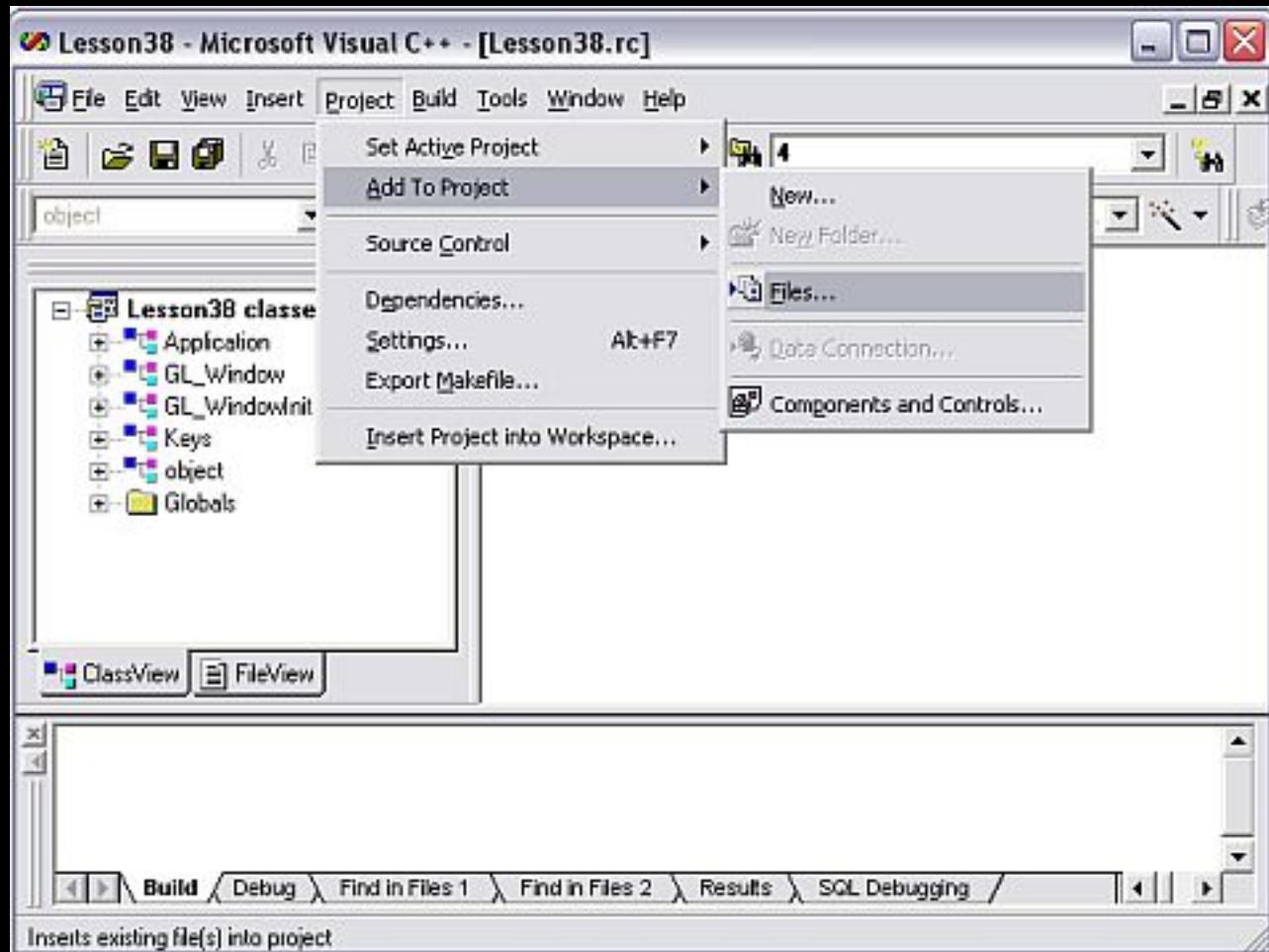


右健单击每个ID，选"属性"，然后重命名，使之与文件名匹配。就像我图片上那样。

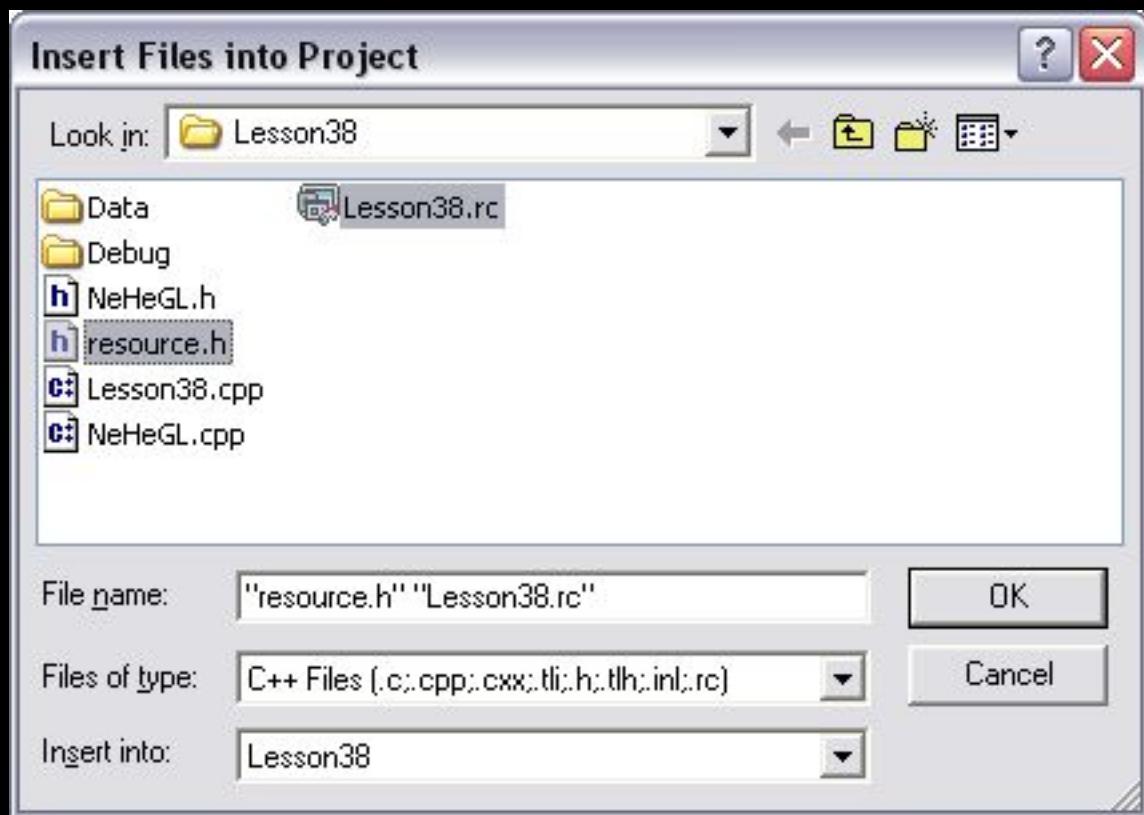


接下来，选“文件-- 全部保存”。你刚刚创建一个新的资源文件，所以Windows会问你取什么名字。你随便拉，也可以叫"lesson38.rc"，然后保存。

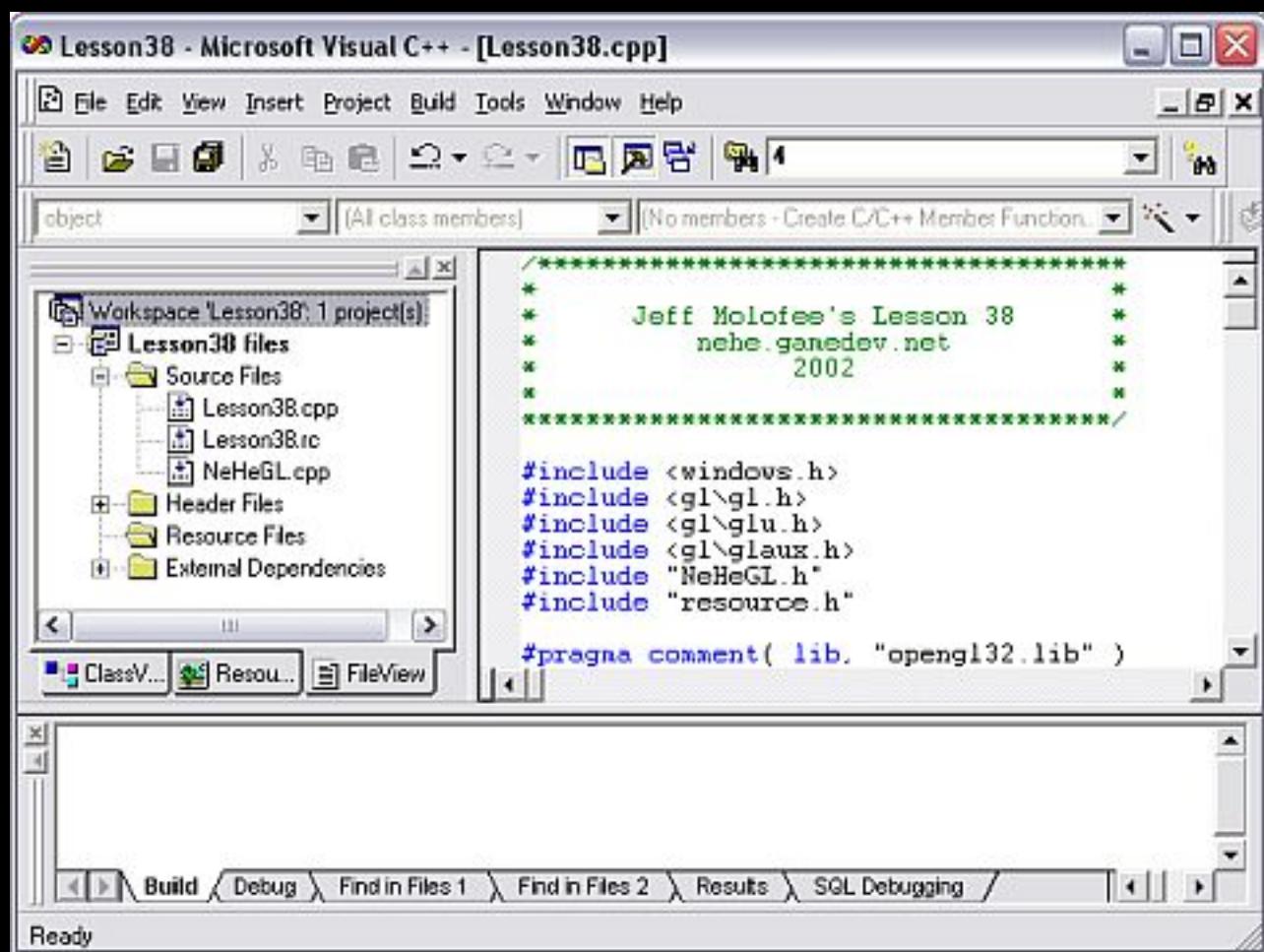
到此为止，你有了一个资源文件，里面全是保存在硬盘上的Bitmap 图形文件，要使用这些文件，你还需要完成一系列步骤。



接下来该把资源文件加到你自己的项目里面了。选“项目-- 添加到项目-- 文件”



选择resource.h文件和资源文件Lesson38.rc (用Ctrl)



最后确认资源文件Lesson38.rc放入RESOURCE FILES文件夹。就像上面图片里那样，点击并拖入RESOURCE FILES文件夹就好了。

移动之后选“文件-- 全部保存”，然后文件部分就好了。好多的图阿:)

然后我们开始code的部分。下面一段最重要的一行是#include "resource.h".没有这行，编译的时候就会有无数未定义变量的错误。resource.h文件定义了资源文件里的对象。所以要从IDB\_BUTTERFLY1里面读取数据的话，最好include这个头文件！

```
#include <windows.h>
#include <gl\gl.h>
#include <gl\glu.h>
#include <gl\glaux.h>
#include "NeHeGL.h"
#include "resource.h" // 资源文件的头文件

#pragma comment( lib, "opengl32.lib" )
#pragma comment( lib, "glu32.lib" )
#pragma comment( lib, "glaux.lib" )

GL_Window* g_window;
Keys* g_keys;
```

下面一段第一行分配三个纹理所需空间，接下来的结构体用于保存关于约50个在屏幕上运动的物体的信息。

tex将跟踪每个物体所用纹理，x是物体的x坐标，y是y坐标，z,z坐标，yi是一个随机数用来控制物体下落速度，

spinz用来控制沿z轴的旋转，spinzi是另一个随机树，记录旋转速度。flap用来控制物体的翅膀（一会在解释这个）随机数fi控制翅膀拍打的速度。

```
// 定义三个保存纹理变量的ID
GLuint texture[3]; // 保存三个纹理

struct object // 定义一个物体
```

```

{
    int tex;                                // 纹理值
    float x;                                 // 位置
    float y;
    float z;
    float yi;                               // 速度
    float spinz;                            // 沿Z轴旋转的角度和速度
    float spinzi;
    float flap;                             // 是否翻转三角形
    float fi;

};

object obj[50];                           // 创建50个物体

```

下面一段代码是物体obj[loop]的初始化，loop从0到49（表示50个物体中的一个）。首先是随机纹理从0到2表示一个随机着色的蝴蝶。x坐标随机的取-17.0f到+17.0f之间的值，y取18.0f，也就是从屏幕的上面一点点开始，这样一开始时看不到物体的。z也是-10.0f到-40.f之间的随机数，spinzi取-1.0f到1.0f之间。flap取翅膀中心位置，为0.0f。最后拍打速度fi和下落速度yi也是随机的。

```

void SetObject(int loop)                  // 循环设置50个物体
{
    obj[loop].tex=rand()%3;                // 纹理
    obj[loop].x=rand()%34-17.0f;           // 位置
    obj[loop].y=18.0f;
    obj[loop].z=-((rand()%30000/1000.0f)+10.0f);
    obj[loop].spinzi=(rand()%10000)/5000.0f-1.0f;   // 旋转
    obj[loop].flap=0.0f;
    obj[loop].fi=0.05f+(rand()%100)/1000.0f;
    obj[loop].yi=0.001f+(rand()%1000)/10000.0f;
}

```

这回该到了最有意思的地方了。从资源文件中读入bitmap，转为纹理。hBMP是指向这个bitmap文件的指针，它将告诉我们的程序从哪里读取数据。BMP是一个bitmap结构体，我们把从资源文件中读取的数据保存在里面。第三行是告诉我们的程序我们将使用哪些ID：IDB\_BUTTERFLY1, IDB\_BUTTERFLY2, IDB\_BUTTERFLY3。要用更多的图像的话，只需增加资源文件中的图像，并在Texture[]中增加新的ID。

```
void LoadGLTextures() // 资源文件中读入bitmap，转为纹理
{
    HBITMAP hBMP; // 位图句柄
    BITMAP BMP; // 位图结构

    // 纹理句柄
    byte Texture[]={ IDB_BUTTERFLY1, IDB_BUTTERFLY2, IDB_BUTTERFLY3 };
```

下面一行使用sizeof(Texture)来计算要创建多少个纹理。我们有3个ID，也就是3个纹理。

```
glGenTextures(sizeof(Texture), &texture[0]); // 创建三个纹理
for (int loop=0; loop<sizeof(Texture); loop++) // 循环载入所有的位图
{
```

LoadImage需要如下参数：GetModuleHandle(NULL)-指向实例的句柄，MAKEINTRESOURCE(Texture[loop])-把Texture[loop]从整型转为一个资源值，也就是要读的图形文件。IMAGE\_BITMAP-告诉我们要读的是一个bitmap文件。接下来两个参数(0,0)是读入图像的高度和宽度像素数，使用默认大小就设为0。最后一个参数(LR\_CREATEDIBSECTION)返回DIB section bitmap??这是一个没有保存颜色信息的bitmap。也正是我们需要的。hBMP 指向从LoadImage()读入的bitmap数据。

```
hBMP=(HBITMAP)LoadImage(GetModuleHandle(NULL),MAKEINTRESOURCE(Texture[loop]), IMAGE_BITMAP, 0, 0, LR_CREATEDIBSECTION);
```

检查指针hBMP是否有效，即指向有用数据。如果没有指向任何数据，也可以弹出错误提示。

如果数据存在，用GetObject()从hBMP取得数据(sizeof(BMP))并存储在BMP中。

`glPixelStorei`告诉OpenGL这些数据是以word alignments存储的，也就是每像素4字节。

绑定纹理，设置滤波方式为GL\_LINEAR\_MIPMAP\_LINEAR(又好又快光滑)，然后生成纹理。

注意到我们使用BMP.bmWidth和BMP.bmHeight获取图像的高度和宽度。并用GL\_BGR\_EXT交换红蓝，实际使用的资源数据是从BMP.bmBits中取得的

◦ 最后删除bitmap对象，释放所有与之相联系的系统资源空间。

```
if (hBMP) // 位图是否存在
{
    // 存在
    GetObject(hBMP,sizeof(BMP), &BMP); // 获得位图
    glPixelStorei(GL_UNPACK_ALIGNMENT,4); // 以四字节方式对其内存
    glBindTexture(GL_TEXTURE_2D, texture[loop]); // 绑定位图
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR); // 设过滤器
    glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,
NEAR_MIPMAP_LINEAR);
    // 创建纹理
    gluBuild2DMipmaps(GL_TEXTURE_2D, 3, BMP.bmWidth, BMP.bmHeight,
R_EXT, GL_UNSIGNED_BYTE, BMP.bmBits);
    DeleteObject(hBMP); // 删除位图对象
}
```

init code没有什么新鲜的，只是增加了LoadGLTextures()调用上面的code。清屏的颜色是黑色，不进行深度检测，这样比较快。启用纹理映射和混色效果。

```
BOOL Initialize (GL_Window* window, Keys* keys) // 初始化
{
    g_window    = window;
    g_keys      = keys;
```

```

LoadGLTextures();                                //载入纹理

glClearColor (0.0f, 0.0f, 0.0f, 0.5f);          // 设置背景
glClearDepth (1.0f);
glDepthFunc (GL_LEQUAL);
glDisable(GL_DEPTH_TEST);                      // 启用深度测试
glShadeModel (GL_SMOOTH);
glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
glEnable(GL_TEXTURE_2D);                        // 启用2D纹理
glBlendFunc(GL_ONE,GL_SRC_ALPHA);               // 使用混合
glEnable(GL_BLEND);

```

初始化所有的物体

```

for (int loop=0; loop<50; loop++)
{
    SetObject(loop);
}

return TRUE;                                     // 成功返回
}

void Deinitialize (void)
{
}

void Update (DWORD milliseconds)                // 更新,执行动画
{
    if (g_keys->keyDown [VK_ESCAPE] == TRUE)      // 按ESC退出
    {
        TerminateApplication (g_window);
    }

    if (g_keys->keyDown [VK_F1] == TRUE)           // 按F1切换显示模式
    {
        ToggleFullscreen (g_window);
    }
}

```

接下来看看绘制代码。在这部分我将讲解如何用尽可能简单的方式将一个图像映到两个三角形上。有些人认为有理由相信，一个图像到三角形上的单一映射是不可能的。

实际上，你可以轻而易举地将图像映到任何形状的区域内。使得图像与边界匹配或者完全不考虑形式。根本没关系的。（译者：我想作者的意思是，从长方形到三角形的解析影射是不存在的，但不考虑那么多的话，任意形状之间的连续影射总是可以存在的。他说的使纹理与边界匹配，大概是指某一种参数化的方法，简单地说使得扭曲最小。）

首先清屏，循环润色50个蝴蝶对象。

```
void Draw (void)                                // 绘制场景
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

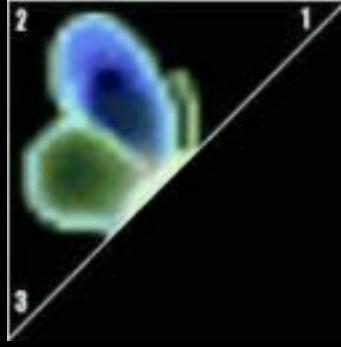
    for (int loop=0; loop<50; loop++)
    {
```

调用glLoadIdentity()重置投影矩阵，然后选择对象的纹理。用glTranslatef()为蝴蝶定位，然后沿x轴旋转45度。使之向观众微略倾斜，这样比较有立体感。最后沿z轴旋转，蝴蝶就旋转下落了。

```
    glLoadIdentity();                            // 重置矩阵
    glBindTexture(GL_TEXTURE_2D, texture[obj[loop].tex]);      // 绑定纹理
    glTranslatef(obj[loop].x,obj[loop].y,obj[loop].z);          // 绘制物体
    glRotatef(45.0f,1.0f,0.0f,0.0f);
    glRotatef((obj[loop].spinZ),0.0f,0.0f,1.0f);
```

其实到三角形上的映射和到方形上并没有很大区别。只是你只有三个定点，要小心一点。

下面的code中，我们将会值第一个三角形。从一个设想的方形的右上角开始，到左上角，再到左下角。润色的结果像下面这样：

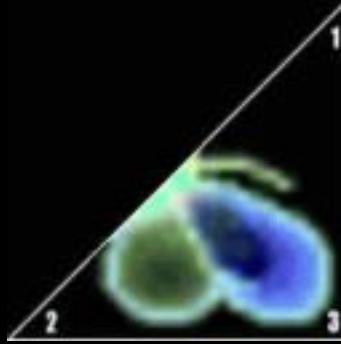


注意半个蝴蝶出现了。另外半个出现在第二个三角形里。同样地将三个纹理坐标与顶点坐标非别对应，这给出充分的信息定义一个三角形上的映射。

```
glBegin(GL_TRIANGLES);

glTexCoord2f(1.0f,1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
glTexCoord2f(0.0f,1.0f); glVertex3f(-1.0f, 1.0f, obj[loop].flap);
glTexCoord2f(0.0f,0.0f); glVertex3f(-1.0f,-1.0f, 0.0f);
```

下面的code润色另一半。同上，只是我们的三角变成了从右上到左下，再到右下。



第一个三角形的第二点和第二个三角形的第三点（也就是翅膀的尖端）在z方向往复运动（即 $z=-1.0f$ 和 $1.0f$ 之间），两个三角形沿着蝴蝶的身体折叠起来，产生拍打的效果，简易可行。

```

glTexCoord2f(1.0f,1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
glTexCoord2f(0.0f,0.0f); glVertex3f(-1.0f,-1.0f, 0.0f);
glTexCoord2f(1.0f,0.0f); glVertex3f( 1.0f,-1.0f, obj[loop].flap);

glEnd();

```

下面一段通过从obj[loop].y中递减obj[loop].yi使蝴蝶自上而下运动。spinz值递增spinzi  
(可正可负) flap递增fi.fi的正负取决于翅膀向上还是向下运动。

```

//移动，选择图像
obj[loop].y-=obj[loop].yi;
obj[loop].spinz+=obj[loop].spinzi;
obj[loop].flap+=obj[loop].fi;

```

当蝴蝶向下运行时，需要检查是否越出屏幕，如果是，就调用SetObject(loop)来给蝴蝶赋新的纹理，新下落速度等。

```

if (obj[loop].y<-18.0f)                                //判断是否超出了屏幕，如果是重置它
{
    SetObject(loop);
}

```

翅膀拍打的时候，还要检查flap是否小于-1.0f或大于1.0f，如果是，令fi=-fi，以改变运动方向。Sleep(15)是用来减缓运行速度，每帧15毫秒。在我朋友的机器上，这让蝴蝶疯狂的飞舞。不过我懒得改了：)

```

if ((obj[loop].flap>1.0f) || (obj[loop].flap<-1.0f))
{
    obj[loop].fi=-obj[loop].fi;
}

```

```
Sleep(15);  
glFlush();  
}
```

希望你在这一课学的开心。也希望通过这一课，从资源文件里读取纹理，和三角形映射的过程变得比较容易理解。我花五分钟又冲读了一遍，感觉还好。如果你还有什么问题，尽管问。我希望我的讲义尽可能好，因此期待您的任何回应。

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

#### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

#### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。



感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

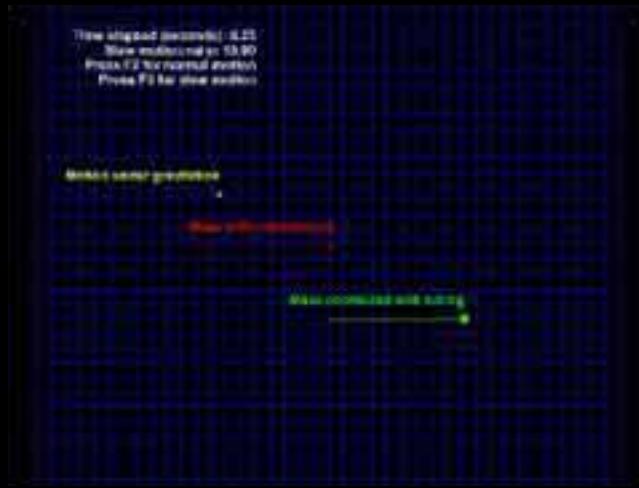
感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第37课

第39课 >



## 第39课



### 物理模拟简介:

还记得高中的物理吧，直线运动，自由落体运动，弹簧。在这一课里，我们将创造这一切。

### 物理模拟介绍

如果你很熟悉物理规律，并且想实现它，这篇文章很适合你。

在这篇教程里，你会创建一个非常简单的物理引擎，我们将创建以下类：

#### 内容:

##### 位置类

\* class Vector3D

--- 用来记录物体的三维坐标的类  
>

##### 力和运动

\* class Mass

--- 表示一个物体的物理属性  
>

##### 模拟类

```

* class Simulation           --- 模拟物理规律
>                               |
模拟匀速运动

* class ConstantVelocity : public Simulation    --- 模拟匀速运动
>                               |
模拟在力的作用下运动

* class MotionUnderGravitation : public Simulation    --- 模拟在引力的作用下运动
>                               |
* class MassConnectedWithSpring : public Simulation    --- 模拟在弹簧的作用下运动
>

```

```

class Mass
{
public:
    float m;                      // 质量
    Vector3D pos;                 // 位置
    Vector3D vel;                 // 速度
    Vector3D force;                // 力

    Mass(float m)                  // 构造函数
    {
        this->m = m;
    }
}

```

...  
下面的代码给物体增加一个力，在初始时这个力为0

```

void applyForce(Vector3D force)
{
    this->force += force;          // 增加一个力
}

void init()                      // 初始时设为0

```

```
{  
    force.x = 0;  
    force.y = 0;  
    force.z = 0;  
}  
  
...
```

下面的步骤完成一个模拟：

- 1.设置力
- 2.应用外力
- 3.根据力的时间，计算物体的位置和速度

```
void simulate(float dt)  
{  
    vel += (force / m) * dt; // 更新速度  
  
    pos += vel * dt; // 更新位置  
}
```

模拟类怎样运作：

在一个物理模拟中，我们按以下规律进行模拟，设置力，更新物体的位置和速度，按时间一次又一次的进行模拟。下面是它的实现代码：

```
class Simulation  
{  
public:  
    int numOfMasses; // 物体的个数  
    Mass** masses; // 指向物体结构的指针  
  
    Simulation(int numOfMasses, float m) // 构造函数
```

```
{  
    this->numOfMasses = numOfMasses;  
  
    masses = new Mass*[numOfMasses];  
  
    for (int a = 0; a < numOfMasses; ++a)  
        masses[a] = new Mass(m);  
}  
  
virtual void release() // 释放所有的物体  
{  
    for (int a = 0; a < numOfMasses; ++a)  
    {  
        delete(masses[a]);  
        masses[a] = NULL;  
    }  
  
    delete(masses);  
    masses = NULL;  
}  
  
Mass* getMass(int index)  
{  
    if (index < 0 || index >= numOfMasses) // 返回第i个物体  
        return NULL;  
  
    return masses[index];  
}  
  
...  
  
(class Simulation continued)  
  
virtual void init() // 初始化所有的物体  
{  
    for (int a = 0; a < numOfMasses; ++a)  
        masses[a]->init();  
}  
  
virtual void solve() //虚函数，在具体的应用中设置各个施加给各个物  
体的力  
{
```

```
}

virtual void simulate(float dt)          //让所有的物体模拟一步
{
    for (int a = 0; a < numOfMasses; ++a)
        masses[a]->simulate(dt);
}

...
```

整个模拟的部分被封装到下面的函数中

(class Simulation continued)

```
virtual void operate(float dt)           // 完整的模拟过程
{
    init();                                // 设置力为0
    solve();                               // 应用力
    simulate();                            // 模拟
}
};
```

现在我们已经有了一个简单的物理模拟引擎了，它包含有物体和模拟两个类，下面我们基于它们创建三个具体的模拟对象：

1. 具有恒定速度的物体
2. 具有恒定加速度的物体
3. 具有与距离成反比的力的物体

在程序中控制一个模拟对象：

在我们写一个具体的模拟类之前，让我们看看如何在程序中模拟一个对象，在这个教程里，模拟引擎和操作模拟的程序在两个文件里，在程序中我们使用如下的函数，操作模拟：

```
void Update (DWORD milliseconds)
```

// 执行模拟

这个函数在每一帧的开始更新 , 参数为相隔的时间。

```
void Update (DWORD milliseconds)
```

```
{
```

```
...
```

```
...
```

```
...
```

```
float dt = milliseconds / 1000.0f; // 转化为秒
```

```
dt /= slowMotionRatio; // 除以模拟系数
```

```
timeElapsed += dt; // 更新流失的时间
```

```
...
```

在下面的代码中 , 我们定义一个处理间隔 , 没隔这么长时间 , 让物理引擎模拟一次。

```
...
```

```
float maxPossible_dt = 0.1f; // 设置模拟间隔
```

```
int numOflterations = (int)(dt / maxPossible_dt) + 1; //计算在流失的时间里模拟的次数
```

```
if (numOflterations != 0)
```

```
    dt = dt / numOflterations;
```

```
for (int a = 0; a < numOflterations; ++a) // 模拟它们
```

```
{
```

```
    constantVelocity->operate(dt);
```

```
    motionUnderGravitation->operate(dt);
```

```
    massConnectedWithSpring->operate(dt);
```

```
}
```

}

下面让我们来写着两个具体的模拟类:

### 1. 具有恒定速度的物体

\* class ConstantVelocity : public Simulation ---> 模拟一个匀速运动的物体

```
class ConstantVelocity : public Simulation
{
public:
    ConstantVelocity() : Simulation(1, 1.0f)
    {
        masses[0]->pos = Vector3D(0.0f, 0.0f, 0.0f);           // 初始位置为0
        masses[0]->vel = Vector3D(1.0f, 0.0f, 0.0f);           // 向右运动
    }
};
```

下面我们来创建一个具有恒定加速的物体 :

```
class MotionUnderGravitation : public Simulation
{
    Vector3D gravitation;                                // 加速度

    MotionUnderGravitation(Vector3D gravitation) : Simulation(1, 1.0f)           // 构造函数
    {
        this->gravitation = gravitation;                // 设置加速度
        masses[0]->pos = Vector3D(-10.0f, 0.0f, 0.0f);      // 设置位置为左边-10处
        masses[0]->vel = Vector3D(10.0f, 15.0f, 0.0f);      // 设置速度为右上
    }

    ...
}
```

下面的函数设置施加给物体的力

```
virtual void solve() // 设置当前的力
{
    for (int a = 0; a < numOfMasses; ++a)
        masses[a]->applyForce(gravitation * masses[a]->m);
}
```

下面的类创建一个受到与距离成正比的力的物体：

```
class MassConnectedWithSpring : public Simulation
{
public:
    float springConstant; // 弹性系数
    Vector3D connectionPos; // 连接方向

    MassConnectedWithSpring(float springConstant) : Simulation(1, 1.0f) // 构造函数
    {
        this->springConstant = springConstant;

        connectionPos = Vector3D(0.0f, -5.0f, 0.0f);

        masses[0]->pos = connectionPos + Vector3D(10.0f, 0.0f, 0.0f);
        masses[0]->vel = Vector3D(0.0f, 0.0f, 0.0f);
    }

    ...
}
```

下面的函数设置当前物体所受到的力：

```
virtual void solve() // 设置当前的力
{
```

```
for (int a = 0; a < numOfMasses; ++a)
{
    Vector3D springVector = masses[a]->pos - connectionPos;
    masses[a]->applyForce(-springVector * springConstant);
}
```

好了上面就是一个简单的物理模拟，希望你能喜欢：）



### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

### 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第38课

第40课 >

## 第40课



绳子的模拟:

怎样模拟一根绳子呢，把它想象成一个个紧密排列的点，怎么样有了思路了吧，在这一课你将学会怎样建模，简单吧，你能模拟更多。

### 绳索模拟

在这个教程里我们将模拟一段绳索，我们是在39课的基础上进行的。

在物理模拟中，我们必须设置各个物理量，就像它们在自然界中的行为一样。模拟中的运动并不一定和自然界相同，我们使用的运动模型，必须和我们需要模拟的目的有关，目的决定了它的精确度。要知道我们的目标不是模拟原子和分子，也不是模拟成千上万的粒子系。首先我们需要确定我们模拟的目标，才能创建我们的物理模型。它和下面内容相关：

1. 运动的数学表示
2. 执行模拟的计算机的速度

#### 1. 运动的数学表示:

这个问题决定了我们使用何种数学方程来模拟运动，使用经典力学还是量子力学。

#### 2. 执行模拟的计算机的速度:

计算机的速度决定了我们可以模拟的精度。

## 设计绳索的物理模型:

我们在经典力学和高于500Mhz的计算机上模拟这个问题。首先我们需要设定需要的精度，我们使用一系列互相用弹簧连接的质点来模拟绳索，精度决定了我们用多少个点来模拟，当然越多越精确。在下面我决定用50或100个点来模拟绳子一段3或4m长的绳子，换句话说，我们的模拟精度就是3到8厘米。

## 设计运动模型:

在绳子中，施加给各个质点的力来自于自身的质量和相连的内力（参见大学里的普通力学）。如下我们用“O”表示质点，“—”表示连接质点的弹簧。

O----O----O----O  
1 2 3 4

弹簧的力学公式如下：

$$\text{力} = -k * x$$

k: 弹性系数

x: 相距平衡位置的位移

上面的公式说明，如果相邻点的距离为平衡距离，那么它们不受到任何力的作用。如果我们设置平衡位置为5cm，那么100个点的绳子长5m。如果相连质点之间的位置小于5cm，它们受到排斥力。

上面的公式只是一个基础，现在我们可以加上摩擦力，如果没有这项，那么绳子将永远动下去。

## 弹簧类:

这个类包含相连接的两个物体，它们之间具有作用力。

```
class Spring
{
public:
    Mass* mass1;                      // 质点1
    Mass* mass2;                      // 质点2
    float springConstant;              // 弹性系数
    float springLength;                // 弹簧长度
```

```
float frictionConstant; //摩擦系数

Spring(Mass* mass1, Mass* mass2,
// 构造函数
float springConstant, float springLength, float frictionConstant)
{
    this->springConstant = springConstant;
    this->springLength = springLength;
    this->frictionConstant = frictionConstant;

    this->mass1 = mass1;
    this->mass2 = mass2;
}

void solve() // 计算各个物体的受力
{
    Vector3D springVector = mass1->pos - mass2->pos;

    float r = springVector.length(); // 计算两个物体之间的距离

    Vector3D force;

    if (r != 0) // 计算力
        force += -(springVector / r) * (r - springLength) * springConstant;
    ...

    force += -(mass1->vel - mass2->vel) * frictionConstant; // 加上摩擦力
    mass1->applyForce(force); // 给物体1施加力
    mass2->applyForce(-force); // 给物体2施加力
}
```

下面我们把绳子钉在墙上，所以我们的模拟就多了一个万有引力，空气摩擦力。万有引力的公式如下：

$$\text{力} = (\text{重力加速度}) * \text{质量}$$

万有引力会作用在每一个质点上，地面也会给每个物体一个作用力。在我们的模型中将考虑绳子和地面之间的接触，地面给绳子向上的力，并提供摩擦力。

### 设置模拟的初始值

现在我们已经设置好模拟环境了，长度单位是m，时间单位是秒，质量单位是kg。

为了设置初始值，我们必须提供供模拟开始的参数。我们定义一下参数：

1. 重力加速度: 9.81 m/s/s 垂直向下
2. 质点个数: 80
3. 相连质点的距离: 5 cm (0.05 meters)
4. 质量: 50 克(0.05 kg)
5. 绳子开始处于垂直状态

下面计算绳子受到的力

$$f = (\text{绳子质量}) * (\text{重力加速度}) = (4 \text{ kg}) * (9.81) \approx 40 \text{ N}$$

弹簧必须平衡这个力 40 N，它伸长1cm，计算弹性系数:

$$\text{合力} = -k * x = -k * 0.01 \text{ m}$$

合力应该为0：

$$40 \text{ N} + (-k * 0.01 \text{ meters}) = 0$$

弹性系数 k 为:

$$k = 4000 \text{ N / m}$$

设置弹簧的摩擦系数:

$$\text{springFrictionConstant} = 0.2 \text{ N/(m/s)}$$

下面我们看看这个绳索类：

1. virtual void init()       $\cdots$  重置力  
    >

- 2. virtual void solve()      --- 计算各个质点的力  
                        >
- 3. virtual void simulate(float dt) --- 模拟一次  
                        >
- 4. virtual void operate(float dt) --- 执行一次操作  
                        >

绳索类如下所示：

```
class RopeSimulation : public Simulation                                    //绳索类
{
public:
    Spring** springs;                                                        // 弹簧类结构的数组的指针

    Vector3D gravitation;                                                // 万有引力

    Vector3D ropeConnectionPos;                                        // 绳索的连接点

    Vector3D ropeConnectionVel;                                        //连接点的速度，我们使用这个移动绳子

    float groundRepulsionConstant;                                    //地面的反作用力

    float groundFrictionConstant;                                    //地面的摩擦系数

    float groundAbsorptionConstant;                                    //地面的缓冲力

    float groundHeight;                                                //地面高度

    float airFrictionConstant;                                        //空气的摩擦系数
```

下面是它的构造函数

```
RopeSimulation(
    int numOfMasses,
    float m,
    float springConstant,
```

```

float springLength,
float springFrictionConstant,
Vector3D gravitation,
float airFrictionConstant,
float groundRepulsionConstant,
float groundFrictionConstant,
float groundAbsorptionConstant,
float groundHeight
) : Simulation(numOfMasses, m)
{
    this->gravitation = gravitation;

    this->airFrictionConstant = airFrictionConstant;

    this->groundFrictionConstant = groundFrictionConstant;
    this->groundRepulsionConstant = groundRepulsionConstant;
    this->groundAbsorptionConstant = groundAbsorptionConstant;
    this->groundHeight = groundHeight;

    for (int a = 0; a < numOfMasses; ++a)          // 设置质点位置
    {
        masses[a]->pos.x = a * springLength;
        masses[a]->pos.y = 0;
        masses[a]->pos.z = 0;
    }

    springs = new Spring*[numOfMasses - 1];

    for (a = 0; a < numOfMasses - 1; ++a)          // 创建各个质点之间的模拟弹簧
    {
        springs[a] = new Spring(masses[a], masses[a + 1],
                               springConstant, springLength, springFrictionConstant);
    }
}

```

计算施加给各个质点的力

```

void solve()          // 计算施加给各个质点的力
{

```

```

for (int a = 0; a < numOfMasses - 1; ++a)           // 弹簧施加给各个物体的力
{
    springs[a]->solve();
}

for (a = 0; a < numOfMasses; ++a)                  // 计算各个物体受到的其它的力
{
    masses[a]->applyForce(gravitation * masses[a]->m); // 万有引力
    // 空气的摩擦力
    masses[a]->applyForce(-masses[a]->vel * airFrictionConstant);

    if (masses[a]->pos.y < groundHeight)           // 计算地面对质点的作用
    {
        Vector3D v;

        v = masses[a]->vel;                         // 返回速度
        v.y = 0;                                     // y方向的速度为0

        // 计算地面给质点的力
        masses[a]->applyForce(-v * groundFrictionConstant);

        v = masses[a]->vel;
        v.x = 0;
        v.z = 0;

        if (v.y < 0)                                // 计算地面的缓冲力

            masses[a]->applyForce(-v * groundAbsorptionConstant);

        // 计算地面的反作用力
        Vector3D force = Vector3D(0, groundRepulsionConstant, 0) *
            (groundHeight - masses[a]->pos.y);

        masses[a]->applyForce(force);               // 施加地面对质点的力
    }
}
}

```

下面的代码完成整个模拟过程

```

void simulate(float dt)           // 模拟一次
{
    Simulation::simulate(dt);     // 调用基类的模拟函数

    ropeConnectionPos += ropeConnectionVel * dt;      // 计算绳子的连接点

    if (ropeConnectionPos.y < groundHeight)
    {
        ropeConnectionPos.y = groundHeight;
        ropeConnectionVel.y = 0;
    }

    masses[0]->pos = ropeConnectionPos;             // 更新绳子的连接点和速度
    masses[0]->vel = ropeConnectionVel;
}

void setRopeConnectionVel(Vector3D ropeConnectionVel)
{
    this->ropeConnectionVel = ropeConnectionVel;
}

```

有了上面的类，我们可以很方便的模拟绳子，代码如下：

```

RopeSimulation* ropeSimulation =
new RopeSimulation(
    80,                                // 80 质点
    0.05f,                             // 每个质点50g
    10000.0f,                          // 弹性系数
    0.05f,                             // 质点之间的距离
    0.2f,                              // 弹簧的内摩擦力
    Vector3D(0, -9.81f, 0),           // 万有引力
    0.02f,                            // 空气摩擦力
    100.0f,                           // 地面反作用系数
    0.2f,                             // 地面摩擦系数
    2.0f,                            // 地面缓冲系数
    -1.5f);                           // 地面高度

```

下面的代码在程序中执行绳子的模拟

```
float dt = milliseconds / 1000.0f; // 经过的秒数  
float maxPossible_dt = 0.002f; // 模拟间隔  
  
int numIterations = (int)(dt / maxPossible_dt) + 1; // 模拟次数  
if (numIterations != 0)  
    dt = dt / numIterations;  
  
for (int a = 0; a < numIterations; ++a) // 执行模拟  
    ropeSimulation->operate(dt);
```

我相信这一个教会了你很多，从最开始的模型的建立，到完成最后的代码。有了这个基础，相信你会创造出很多更有意思的代码！

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

#### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自



己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

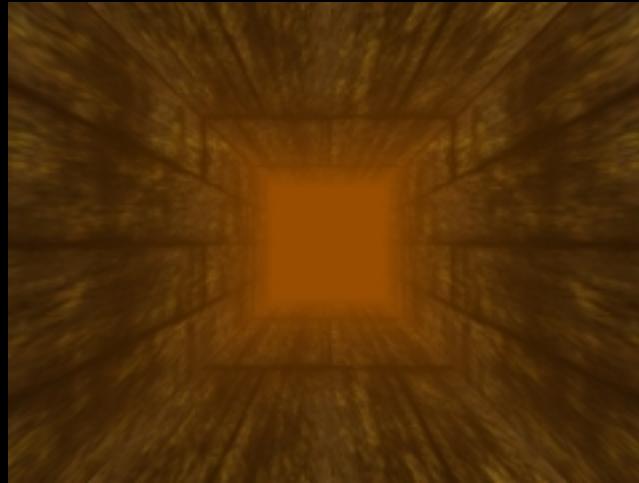
感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第39课

第41课 &gt;



## 第41课



体积雾气

把雾坐标绑定到顶点，你可以在雾中漫游，体验一下吧。

这一课我们将介绍体积雾，为了运行这个程序，你的显卡必须支持扩展"GL\_EXT\_fog\_coord"。

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

#include "NeHeGL.h"

#pragma comment( lib, "opengl32.lib" )
#pragma comment( lib, "glu32.lib" )

GL_Window* g_window;
Keys* g_keys;
```

下面的代码设置雾的颜色和摄像机在Z方向的深度

```
GLfloat fogColor[4] = {0.6f, 0.3f, 0.0f, 1.0f};           // 雾的颜色
GLfloat camz;                                            // 摄像机在Z方向的深度
```

下面变量GL\_FOG\_COORDINATE\_SOURCE\_EXT和GL\_FOG\_COORDINATE\_EXT具有初值，他们在glext.h文件中被定义，这里我们必须感谢Lev Povalahev，它创建了这个文件。如果你想编译你的代码，你必须设置这个值。

为了使用glFogCoordfExt，我们需要定义这个函数的指针，并在程序运行时把它指向显卡中的函数。

// 使用FogCoordfEXT它需要的变量

```
#define GL_FOG_COORDINATE_SOURCE_EXT 0x8450          // 从GLEXT.H得到的值
#define GL_FOG_COORDINATE_EXT      0x8451
```

typedef void (APIENTRY \* PFNGLFOGCOORDFEXTPROC) (GLfloat coord); // 声明雾坐标函数的原形

```
PFNGLFOGCOORDFEXTPROC glFogCoordfEXT = NULL;          // 设置雾坐标函数指针为NULL
```

GLuint texture[1]; // 纹理

Nehe的原文介绍了Ipicture的接口，它不是我们这一课的重点，故我还是使用以前的方法加载纹理。

下面的代码用来检测用户的显卡是否支持EXT\_fog\_coord扩展，这段代码只有在你获得OpenGL渲染描述表后才能调用，否则你将获得一个错误。

首先，我们创建一个字符串，来描述我们需要的扩展。接着我们分配一块内存，用来保存显卡支持的扩展，它可以通过glGetString函数获得。接着我们检测是否含有需要的扩展，如果不存在，则返回false，如存在我们把函数的指针指向这个扩展。

```
int Extension_Init()
{
    char Extension_Name[] = "EXT_fog_coord";

    // 返回扩展字符串
    char* glextstring=(char *)malloc(strlen((char *)glGetString(GL_EXTENSIONS))+1);
    strcpy(glextstring,(char *)glGetString(GL_EXTENSIONS));

    if (!strstr(glextstring,Extension_Name))          // 查找是否有我们想要的扩展
        return FALSE;

    free(glextstring);                                // 释放分配的内存

    // 获得函数的指针
    glFogCoordfEXT = (PFNGLFOGCOORDFEXTPROC) wglGetProcAddress("glFogCoordfEXT");

    return TRUE;
}
```

下面的代码初始化OpenGL，并设置雾气的参数。

```
BOOL Initialize (GL_Window* window, Keys* keys)           // 初始化
{
    g_window      = window;
    g_keys        = keys;

    // 初始化扩展
    if (!Extension_Init())
        return FALSE;

    if (!BuildTexture("data/wall.bmp", texture[0]))          // 创建纹理
        return FALSE;

    glEnable(GL_TEXTURE_2D);
    glClearColor (0.0f, 0.0f, 0.0f, 0.5f);
    glClearDepth (1.0f);
```

```
glDepthFunc(GL_LESS);
 glEnable(GL_DEPTH_TEST);
 glShadeModel(GL_SMOOTH);
 glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
```

下面的代码设置雾气的属性

最后一个设置为雾气基于顶点的坐标，这运行我们把雾气放置在场景中的任意地方。

```
glEnable(GL_FOG);
 glFogi(GL_FOG_MODE, GL_LINEAR);
 glFogfv(GL_FOG_COLOR, fogColor);
 glFogf(GL_FOG_START, 1.0f);
 glFogf(GL_FOG_END, 0.0f);
 glHint(GL_FOG_HINT, GL_NICEST);
 glFogi(GL_FOG_COORDINATE_SOURCE_EXT, GL_FOG_COORDINATE_EXT); //设置雾
```

气是基于顶点的坐标

```
camz = -19.0f;
```

```
return TRUE;
}
```

下面的代码绘制具体的场景

```
void Draw (void)
{
 glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
 glLoadIdentity ();
 glTranslatef(0.0f, 0.0f, camz);
```

下面的代码绘制四边形组成的墙，并设置每个顶点的纹理坐标和雾坐标

```
glBegin(GL_QUADS); //后墙
    glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f(-2.5f, -2.5f, -15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f( 2.5f, -2.5f, -15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f( 2.5f, 2.5f, -15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f(-2.5f, 2.5f, -15.0f);
glEnd();

glBegin(GL_QUADS); // 地面
    glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f(-2.5f, -2.5f, -15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f( 2.5f, -2.5f, -15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f( 2.5f, -2.5f, 15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f(-2.5f, -2.5f, 15.0f);
glEnd();

glBegin(GL_QUADS); // 天花板
    glFogCoordfEXT(0.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f(-2.5f, 2.5f, -15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f( 2.5f, 2.5f, -15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f( 2.5f, 2.5f, 15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f(-2.5f, 2.5f, 15.0f);
glEnd();

glBegin(GL_QUADS); // 右墙
    glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f( 2.5f, -2.5f, 15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f( 2.5f, 2.5f, 15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f( 2.5f, 2.5f, -15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f( 2.5f, -2.5f, -15.0f);
glEnd();

glBegin(GL_QUADS); // 左墙
    glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 0.0f); glVertex3f(-2.5f, -2.5f, 15.0f);
    glFogCoordfEXT(1.0f); glTexCoord2f(0.0f, 1.0f); glVertex3f(-2.5f, 2.5f, 15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(1.0f, 1.0f); glVertex3f(-2.5f, 2.5f, -15.0f);
    glFogCoordfEXT(0.0f); glTexCoord2f(1.0f, 0.0f); glVertex3f(-2.5f, -2.5f, -15.0f);
glEnd();

glFlush ();
}
```

希望你喜欢这一课 , 如果你愿意的话创建更漂亮的体积雾吧。

### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称 , 我的联系方式是zhouwei02@mails.tsinghua.edu.cn , 如果你有任何问题 , 都可以联系我。

### 引子

网络是一个共享的资源 , 但我在自己的学习生涯中浪费大量的时间去搜索可用的资料 , 在现实生活中花费了大量的金钱和时间在书店中寻找资料 , 于是我给自己起了个昵称DancingWind , 其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后 , 我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容 , 大多都来自共享的资源 , 所以我没有资格把它们据为己有 , 或声称自己为这些资源作出了一点贡献。故任何人都可以复制 , 修改 , 重新发表 , 甚至以自己的名义发表 , 我都不会追究 , 但你在做以上事情的时候必须保证内容的完整性 , 给后来的人一个完整的教程。最后 , 任何人不能以这些资料的任何部分 , 谋取任何形式的报酬。

### 发展计划

在国外 , 很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识 , 我很欢迎你与我联系 , 但你必须同意我上面的声明。

### 感谢

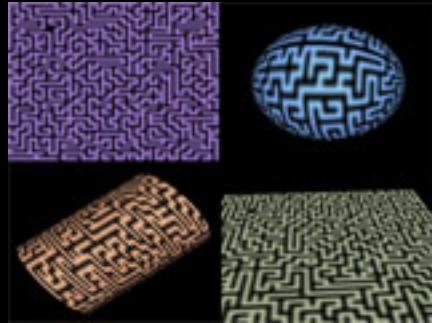
感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹 , 一直以来默默的在精神上和生活中对我的支持 , 她甚至把买衣服的钱都用来给我买书了 , 她真的是我见过的最好的女孩 , 希望我能带给她幸福。





## 第 42课



多重视口

画中画效果，很酷吧。使用视口它变得很简单，但渲染四次可会大大降低你的显示速度哦：）

欢迎来到充满趣味的另一课。这次我将向你展示怎样在单个窗口内显示多个视口。这些视口在窗口模式下能正确的调整大小。其中有两个窗口起用了光照。窗口之一用的是正交投影而其他三个则是透视投影。为了保持教程的趣味性，在本例子中我们同样需要学习迷宫代码，怎么渲染到一张纹理以及怎么得到当前窗口的分辨率。

一旦你明白了本教程，制作分屏游戏以及多视图的3D程序就很简单了。接下来，让我们投入到代码中来吧！！！

你可以利用最近的NeHeGL或者IPicture代码作为主要基本代码。我们需要看的第一个文件就是NeHeGL.cpp,其中有三节代码已经被修改了。我将只列出那些被修改了的代码。

第一个且最重要的被修改了的代码就是ReshapeGL()函数。这是我们设置屏幕(主视口)分辨率的地方。现在所有的主视口设置都在画循环里完成了。因此这儿所有我们能做的就是设置我们的主窗口。

```
void ReshapeGL (int width, int height) // 当窗口移动或者大小改变时重新
调整窗口
{
    glViewport (0, 0, (GLsizei)(width), (GLsizei)(height)); // 重置当前视口
}
```

下一步我们添加一些代码用于监视擦除窗口背景的Windows消息(WM\_ERASEBKGND).如果它被调用，我们截取它并返回0,这样就阻止了窗口背景被擦除，并让我们自己来调整主窗口大小，这样就没有了我们以前常见的那种恼人的闪烁。如果你还不明白我的意思，删掉 case WM\_ERASEBKGND: 和 return 0; 你自己比较就能知道有何不同。

```
LRESULT CALLBACK WindowProc (HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    DWORD      tickCount;           // 保存当前的时间
    __int64    timer;              // 记录时间

    // 返回窗口结构
    GL_Window* window = (GL_Window*)(GetWindowLong (hWnd, GWL_USERDATA));

    switch (uMsg)                 // 处理消息
    {
        case WM_ERASEBKGND:       // 检测Windows是否去擦除背景
            return 0;               // 跳过直接返回
    }
}
```

在WinMain函数中，我们需要修改窗口标题并设置分辨率至1024x768.如果由于某种原因你的显示器不能支持到1024x768,你可以设置低一点的分辨率，但是牺牲了一些细节。

```
window.init.width      = 1024;          // 宽
window.init.height     = 768;           // 高
```

现在该是对lesson42.cpp文件动手术的时候了（主要代码）...  
我们以包含标准头文件和库文件作为开始吧。

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

#include "NeHeGL.h"
```

```
#pragma comment( lib, "opengl32.lib" )
#pragma comment( lib, "glu32.lib" )
```

```
GL_Window* g_window;
Keys* g_keys;
```

然后我们声明一些我们打算在整个程序中都要用到的全局变量。

mx和my纪录了当前所在迷宫中的房间。每个房间都被墙隔开(因此房间都是2个单元大小的部分)。

width和height是用来建立纹理需要的。它也是迷宫的宽和高。让迷宫和贴图的大小一致的原因是使迷宫中的象素和纹理中的象素一一对应。我倾向于把宽和高都设成256,尽管这要花更长的时间来建立迷宫。

如果你的显卡能支持处理大型贴图。可以试着以2次幂增加这个值(256, 512, 1023)。确保这个值不至于太大。如果这个主窗口的宽度有1024个象素，并且每个视口的大小都是主窗口的一半，相应的你应该设置你的贴图宽度也是窗口宽度的一半。如果你使贴图宽度为1024象素，但你的视口大小只有512,空间不足于容纳贴图中所有得象素，这样每两个象素就会重叠在一起。贴图的高度也作同样处理:高度是窗口高度的1/2.当然你还必须四舍五入到2的幂。

```
int mx,my;                                // 循环变量
const width = 128;                          // 迷宫大小
const height = 128;
```

done用来跟踪迷宫是否被建完，后面有这个更详细的解释。

sp用来确认空格键是否处于按下状态。通过按空格键，迷宫就会被重置，然后程序将重新开始画一个新的迷宫。如果我们不去检测空格键是否处于按下状态，迷宫会在空格键按下的瞬间被重置很多次。这个值确保迷宫只被重置一次。

```
BOOL done;                                  // 迷宫是否被建完
BOOL sp;
```

r[4]保存了4个随机的红色分量值，g[4]保存了4个随机的绿色分量值，b[4]保存了4个随机的兰色分量值。这些值赋给各个视口不同的颜色。第一个视口颜色为r[0],g[0],b[0]。请注意每一个颜色都是一个字节的值，而不是常用的浮点值。我这里用字节是因为产生0-255的随机值比产生0.0f-1.0f的浮点值更容易。  
tex\_data指向我们的贴图数据。

```
BYTE r[4], g[4], b[4]; // 随机的颜色
BYTE *tex_data; // 保存纹理数据
```

xrot,yrot和zrot是旋转3d物体用到的变量。

最后，我们声明一个二次曲面物体，这样我们可以用gluCylinder和gluSphere来画圆柱和球体，这比手工绘制这些物体容易多了。

```
GLfloat xrot, yrot, zrot; // 旋转物体
GLUquadricObj *quadric; // 二次几何体对象
```

下面的小段代码设置纹理中位置dmx,dmy的颜色值为纯白色。tex\_data是指向我们的纹理数据的指针。每一个象素都由3字节组成(1字节红色分量，1字节绿色分量，一字节兰色分量)。红色分量的偏移为0，我们要修改的象素的在纹理数据中的偏移为dmx(象素的x坐标)加上dmy(象素y坐标)与贴图宽度的乘积,最后的结果乘3(3字节每象素)。

下面第一行代码设置red(0)颜色分量为255, 第二行设置green(1)颜色分量为255,最后一行设置blue(2)颜色分量为255,最后的结果为在dmx,dmy处的象素颜色为白色。

```
void UpdateTex(int dmx, int dmy) // 更新纹理
{
    tex_data[0+((dmx+(width*dmy))*3)]=255; // 设置颜色为白色
    tex_data[1+((dmx+(width*dmy))*3)]=255;
    tex_data[2+((dmx+(width*dmy))*3)]=255;
}
```

重置有相当多的工作量。它清空纹理，给每一个视口设置随机颜色，删除迷宫中的墙并为迷宫的生成设置新的随机起点。

第一行代码清空tex\_data指向的贴图数据。我们需要清空width(贴图宽) \*height(贴图高)\*3(红，绿，兰)。（代码已经够清楚了，呜呼，干吗要翻译这段？）清空内存空间就是设置所有的字节为0。如果3个颜色分量都清零，那么整个贴图就完全变黑了！

```
void Reset (void)
{
    ZeroMemory(tex_data, width * height *3); // 清空纹理数据
```

现在我们来给每一个视口设置随机的颜色。对于不了解这些的人来说，这里的随机并不是真正那种随机！如果你写了一个简单的程序来打印出10个随机数字，每次你运行程序，你都会得到同样的10个数字。为了使事情（看起来）更加随机，我们可以设置随机数种子。同样的，如果你设置种子为1，你总是会得到同样的结果。然而，如果我们设置srand为开机后当前时钟计数(这可能是任意的数)，我们的程序每次运行都会有不同的结果。

我们有四个视口，因此我们需要从0-3的循环来处理。我们给每一个颜色(red,green,blue)从128-255中间的随机值。要加128的目的是需要更亮的颜色。最小值为0，最大值为255，而128则表示大约有50%的亮度。

```
srand(GetTickCount()); // 初始化随机向量

for (int loop=0; loop<4; loop++) // 循环随机生成颜色
{
    r[loop]=rand()%128+128;
    g[loop]=rand()%128+128;
    b[loop]=rand()%128+128;
}
```

下一步，我们设置一个随机的起点。我们的起点必须是一个房间。在纹理中每两个象素就是一个房间。为确保起点是房间而不是墙，我们在0至贴图宽度一半的范围内挑选一个数，并和2相乘。通过这种方法我们只能得到如0,2,6,8之类的数，也就是说我们总是得到一个随机的房间，决不会着陆到一堵墙上如1,3,5,7,9等等。

```

mx=int(rand()%width/2)*2;
my=int(rand()%height/2)*2;
}
}
```

初始化的第一行代码非常重要。它分配了足够的内存来保存我们的纹理(width\*height\*3). 如果你不分配内存，你很可能使你的系统崩溃。

```

BOOL Initialize (GL_Window* window, Keys* keys)           // 初始化
{
    tex_data=new BYTE[width*height*3];                      // 分配保存纹理的空间

    g_window      = window;
    g_keys        = keys;
```

一 分配完内存，我们就调用Reset()函数，Reset会清空贴图，设置所需颜色，并为迷宫选取随机起点。

一旦所有的东西都设置好了。我们建立我们的初始纹理。前两个纹理参数将纹理坐标截断在 [0,1]范围内，当把一个单独的图像映射到一个物体上时，这种方式可以避免缠绕时人为因素的影响(?本句翻译不爽，请指正). 为了看到CLAMP参数的重要性，可以移掉这两行代码看看。如果没有Clamping,你会注意到在纹理的顶部和右边的细小线条。这些线条的出现是因为线性过滤想使整个纹理平滑，包括纹理边界。如果一个靠近边界的点被画了，在纹理的对边上就会出现一条线。

我们打算用线性过滤来使纹理变的更平滑一点。用什么类型的过滤是由你来决定的。如果它使程序跑起来很慢，那就换成过滤类型为GL\_NEAREST

最后，我们利用tex\_data数据（并没有利用alpha通道）建立了一个二维的RGB纹理。

```
Reset();                                              // 重置纹理贴图
```

```

// 设置纹理参数
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGB, GL_UNSIGNED_BYTE,
tex_data);
```

我们设置用于清空颜色缓冲区的颜色为黑色，清空深度缓冲区的值为1.0f. 设置深度函数为less than或者equal to, 然后激活深度测试。

激活GL\_COLOR\_MATERIAL可以让你在激活光照的情况下用glColor函数给物体上色。这个方法又称为颜色追踪, 常常是性能杀手的glMaterial的代替品。我收到? • 矶都mail问我如何修改物体的颜色...,希望这些信息对这个有帮助！对于那些发email问我为什么纹理的颜色如此怪异或者问纹理颜色受当前glColor影响的人, 请确认一下你没有激活GL\_COLOR\_MATERIAL.

\*多谢James Trotter对GL\_COLOR\_MATERIAL功能的解释。我曾说过它会对你的纹理上色...实际上, 它是对你的物体上色。

最后我们激活2维纹理映射。

```
glClearColor (0.0f, 0.0f, 0.0f, 0.0f);  
glClearDepth (1.0f);
```

```
glDepthFunc (GL_LESS);  
glEnable (GL_DEPTH_TEST);
```

```
glEnable(GL_COLOR_MATERIAL);  
  
glEnable(GL_TEXTURE_2D);
```

下面的代码建立了一个二次曲面物体并得到指向它的指针。一旦我们有这个指针后, 我们设置它的法线类型为平滑类型, 然后要求生成纹理坐标。这样我们的光照才能正确的工作, 并且我们的纹理能自动的映射到二次曲面物体。

```
quadric=gluNewQuadric();  
gluQuadricNormals(quadric, GLU_SMOOTH);  
gluQuadricTexture(quadric, GL_TRUE);
```

Light0被激活, 但是如果我们不激活光照, 它不会起任何作用。Light0是预定义的灯光, 方向指向屏幕内。如果你不喜欢的话, 可以手工自己设置

```
glEnable(GL_LIGHT0);
```

```
return TRUE;
```

```
}
```

只要你分配了内存，记住释放它是很重要的。不管你在全屏切换或者程序退出时，下面的代码都将释放了tex\_data的内存空间。

```
void Deinitialize (void)
{
    delete [] tex_data;
}
```

大部分迷宫建设，以及键盘检测，旋转处理等工作都是在Update()函数中完成的。  
我们需要设置一个变量dir, 用它来表示记录随机的向上，向右，向下或向左值。  
我们需要检测空格键是否被按下，如果是，并且不处于按下状态，我们就重置迷宫。如果按键被释放，我们设置sp为false,这样程序就知道它不再是按下状态。

```
void Update (float milliseconds)                                // 更新各个参数
{
    int dir;                                                 // 保存当前的方向

    if (g_keys->keyDown [VK_ESCAPE])                          // 处理键盘信息
        TerminateApplication (g_window);

    if (g_keys->keyDown [VK_F1])
        ToggleFullscreen (g_window);

    if (g_keys->keyDown [' '] && !sp)
    {
        sp=TRUE;
        Reset();
    }

    if (!g_keys->keyDown [' '])
        sp=FALSE;
```

xrot,yrot和zrot通过和一些小浮点数相乘而随着时间的消逝而增加。这样我们可以让物体绕x轴，y轴和z轴旋转。每个变量都增加不同的值使旋转好看一点

```
xrot+=(float)(milliseconds)*0.02f;
yrot+=(float)(milliseconds)*0.03f;
zrot+=(float)(milliseconds)*0.015f;
```

下面的代码用来检测我们是否画完了迷宫。我们开始设置done值为true, 然后循环检查每一个房间去看是否需要增加一面墙，如果有一间房还有被访问到，我们设置done为false. 如果tex\_data[(x + (width\*y))\*3]的值为0, 我们就明白这个房间还没被访问到，而且没有在里面画一个象素。如果这儿有一个象素,那么它的值为255。我们只需要检查它的颜色红色分量值。因为我们知道这个值只能为0(空)或者255(更新过)

```
done=TRUE; // 循环所有的纹理素，如果为0则表示没有
绘制完所有的迷宫，返回
for (int x=0; x<width; x+=2)
{
    for (int y=0; y<height; y+=2)
    {
        if (tex_data[((x+(width*y))*3]==0)
            done=FALSE;
    }
}
```

检查完所有的房间之后，如果done为true.那么迷宫就算建完了，SetWindowText就会改变窗口的标题。我们改变标题为"迷宫建造完成！"。然后我们停顿5000毫秒使看这个例子的人有时间来看标题栏上的字(如果在全屏状态，他们会看到动画停顿了)。

```
if (done) //如果完成停止五秒后重置
{
    SetWindowText(g_window->hWnd,"Lesson 42: Multiple Viewports... 2003 NeHe Productions...
Maze Complete!");
}
```

```
Sleep(5000);
SetWindowText(g_window->hWnd,"Lesson 42: Multiple Viewports... 2003 NeHe Productions...
Building Maze!");
Reset();
}
```

下面的代码也许让人看着糊涂，但其实并不难懂。我们检查当前房间的右边房间是否被访问过或者是否当前位置的右边是迷宫的右边界（当前房间右边的房间就不存在），同样检查左边的房间是否访问过或者是否达到左边界。其它方向也作如此检查。

如果房间颜色的红色分量的值为255,就表示这个房间已经被访问过了(因为它已经被函数UpdateTex更新过)。如果mx(当前x坐标 ) 小于2, 就表示我们已经到了迷宫最左边不能再左了。

如果往四个方向都不能移动了或以已经到了边界，就给mx和my一个随机值，然后检查这个值对应点是否被访问，如果没有，我们就重新寻找一个新的随机变量，直到该变量对应的单元早已经被访问。因为需要从旧的路径中分叉出新的路径，所以我们必须保持搜索知道发觉有一老的路径可以从那里开始新的路径。

为了使代码尽量简短，我没有打算去检查mx-2是否小于0。如果你想有100%的错误检测，你可以修改这段代码阻止访问不属于当前贴图的内存。

```
// 检测是否走过这里
if (((tex_data[((mx+2)+(width*my))*3]==255) || mx>(width-4)) && ((tex_data[((mx-2)
+(width*my))*3]==255) || mx<2) &&
((tex_data[((mx+(width*(my+2)))*3]==255) || my>(height-4)) && ((tex_data[((mx+(width*
(my-2)))*3]==255) || my<2)))
{
    do
    {
        mx=int(rand()%(width/2))*2;
        my=int(rand()%(height/2))*2;
    }
    while (tex_data[((mx+(width*my))*3]==0);
}
```

下面这行代码赋给dir变量0-3之间的随机值，这个值告诉我们该往右，往上，往左还是往下画迷宫。

在得到随机的方向之后，我们检查dir的值是否为0(往右移)，如果是并且我们不在迷宫的右边界，然后检查当前房间的右边房间，如果没被访问，我们就调用UpdateTex(mx+1,my)在两个房间之间建立一堵墙，然后mx增加2移到新的房间。

```
dir=int(rand()%4); // 随机一个走向

if ((dir==0) && (mx<=(width-4))) // 向右走，更新数据
{
    if (tex_data[((mx+2)+(width*my))*3]==0)
    {
        UpdateTex(mx+1,my);
        mx+=2;
    }
}
```

如果dir的值为1(往下)，并且我们不在迷宫底部，我们检查当前房间的下面房间是否被访问过。如果没被访问过，我们就在两个房间(当前房间和当前房间下面的房间)建立一堵墙。然后my增加2移到新的房间。

```
if ((dir==1) && (my<=(height-4))) // 向下走，更新数据
{
    if (tex_data[((mx+(width*(my+2)))*3]==0)
    {
        UpdateTex(mx,my+1);
        my+=2;
    }
}
```

如果dir的值为2(向左)并且我们不在左边界，我们就检查左边的房间是否被访问，如果没被访问，我们也在两个房间(当前房间和左边的房间)之间建立一堵墙，然后mx减2移到新的房间。

```
if ((dir==2) && (mx>=2)) // 向左走 , 更新数据
{
    if (tex_data[((mx-2)+(width*my))*3]==0)
    {
        UpdateTex(mx-1,my);
        mx-=2;
    }
}
```

如果dir的值为3并且不在迷宫的最顶部，我们检?榈鼻胺考涞纳厦媸欠癖环梦剩 缄•挥校•蛟诹礁龇考?(当前房间和当前房间上面个房间)之间建立一堵墙，然后my增加2移到新的房间。

```
if ((dir==3) && (my>=2)) // 向上走 , 更新数据
{
    if (tex_data[((mx+(width*(my-2)))*3]==0)
    {
        UpdateTex(mx,my-1);
        my-=2;
    }
}
```

移到新的房间后，我们必须标志当前房间为正在访问状态。我们通过调用以当前位置mx, my为参数的UpdateTex()函数来达到这个目的。

```
UpdateTex(mx,my); // 更新纹理
}
```

这段代码我们开始讲一些新的东西...我们必须知道当前窗口的大小以便正确的调整视口的大小。为了的到当前窗口的宽和高，我们需要获取窗口上下左右坐标值。得到这些值后我们通过窗口右边的坐标减去左边的坐标得到宽度值。底部坐标减去顶部坐标得到窗口的高度值。

我们用RECT结构来得到窗口的那些值。RECT保存了一个矩形的坐标。也即矩形的左，右，顶部，底部的坐标。

为获取窗口的屏幕坐标，我们用GetClientRect()函数。我们传进去的第一个参数是当前窗口的句柄。第二个参数是一个结构用于保存返回的窗口位置信息.

```
void Draw (void)                                // 绘制
{
    RECT rect;                                  // 保存长方形坐标

    GetClientRect(g_window->hWnd, &rect);        // 获得窗口大小
    int window_width=rect.right-rect.left;
    int window_height=rect.bottom-rect.top;
```

我们在每一帧都需要更新纹理并且要在映射纹理之前更新。更新纹理最快的方法是用命令glTexSubImage2D(). 它能把内存中的纹理的全部或部分和屏幕中的物体建立映射。下面的代码我们表明用的??2维纹理，纹理细节级别为0，没有x方向(0)或y方向(0)的偏移，我们需要利用整张纹理的每一部分，图像为GL\_RGB类型，对应的数据类型为GL\_UNSIGNED\_BYTE. tex\_data是我们需要映射的具体数据。

这是一个非常快的不用重建纹理而更新纹理的方法。同样需要注意的是这个命令不会为你建立一个纹理。你必须在更新纹理前把纹理建立好。

// 设置更新的纹理

```
glTexSubImage2D(GL_TEXTURE_2D, 0, 0, 0, width, height, GL_RGB, GL_UNSIGNED_BYTE,
tex_data);
```

这行代码非常重要，它将清空整个屏幕。

.....  
一次性清空整个屏幕，然后在画每一个视口前清空它们的深度存非常重要。

```
glClear (GL_COLOR_BUFFER_BIT);
```

现在是主画循环。我们要画四个视口，所以建立了一个0到3的循环。

首先要做的事是设置用glColor3ub(r,g,b)设置当前视口的颜色。这对某些人来说不太熟悉，它跟glColor3f(r,g,b)几乎一样但是用无符号字节代替浮点数为参数。记住早些时候我说过参省一个0-255的随机颜色值会更容易。好在已经有了该命令设置正确颜色所需要的值。

glColor3f(0.5f,0.5f,0.5f)是指颜色中红，绿，蓝具有50%的亮度值。glColor3ub(127,127,127)同样也表示同样的意思。

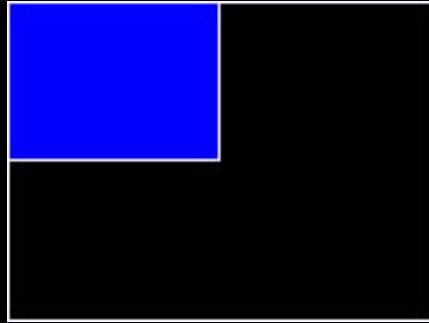
如果loop的值为0,我们将选择r[0],b[0],b[[0]]，如果loop指为1, 我们选用r[1],g[1],b[1]. 这样，每个场景都有自个的随机颜色。

```
for (int loop=0; loop<4; loop++) // 循环绘制4个视口
{
    glColor3ub(r[loop],g[loop],b[loop]);
```

在画之前首先要做的是设置当前视口，如果loop值为0,我们画第一个视口。我们想把第一个视口放在屏幕的左半部分(0),并且在屏幕的上半部分(window\_height/2).视口的宽度为当前主窗口的一半(window\_width/2), 高度也为主窗口高度的一半(window\_height/2).

如果主窗口为1024x768, 结果就是一个起点坐标为0,384,宽512，高384的视口。

这个视口看起来象下面这张图



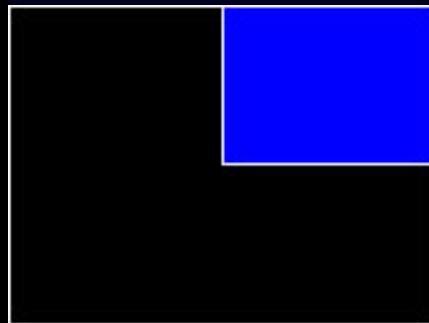
设置完视口后，我们选择当前矩阵为投影矩阵，重置它并设置为2D平行投影视图。我们需要以平行投影视图来填充整个视口，因此我们给左边的值为0,右边的值为window\_width/2(跟视口一样)，同样给底部的值赋为window\_height/2，顶部的值为0. 这样给了视口同样的高度。

这个平行投影视图的左上角的坐标为0,0,右下角坐标为window\_width/2,window\_height/2.

```
if (loop==0) // 绘制左上角的视口
{
    // 设置视口区域
    glViewport (0, window_height/2, window_width/2, window_height/2);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluOrtho2D(0, window_width/2, window_height/2, 0);
}
```

如果loop的值为1, 我们是在画第二个视口了。它在屏幕的右上部分。宽度和高度都跟前一个视图一样。唯一不同的是glViewport()函数的第一个参数为window\_width/2.这告诉程序视口起点是从窗口左起一半的地方。

第二个视口看起来象下面这样:

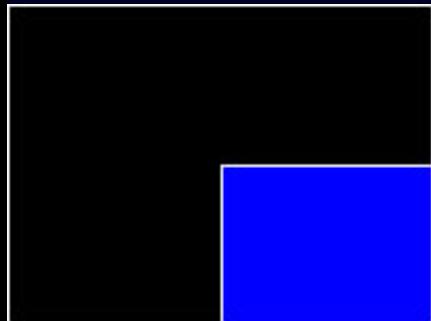


同样的，我们设置当前矩阵为投影矩阵并重置它。但这次我们设置透视投影参数为FOV为45度，并且近截面值为0.1f，远截面值为500.0f

```
if (loop==1) // 绘制右上角视口
{
    glViewport (window_width/2, window_height/2, window_width/2, window_height/2);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective( 45.0, (GLfloat)(width)/(GLfloat)(height), 0.1f, 500.0 );
}
```

如果loop值为2,我们画第三个视口。它将在主窗口的右下部分。宽度和高度与第二个视口一样。跟第二个视口不同的是glViewport()函数的第二个参数为0.这告诉程序我们想让视口位于主窗口的右下部分。

第三个视口看起来如下:

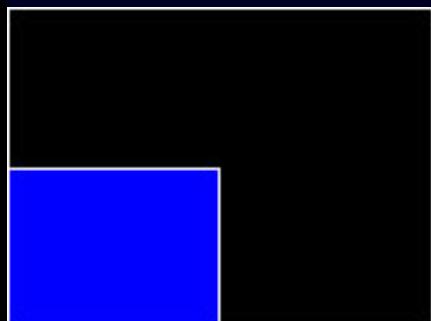


透视视图的设置同第二个视图。

```
if (loop==2) // 绘制右下角视口
{
    glViewport (window_width/2, 0, window_width/2, window_height/2);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective( 45.0, (GLfloat)(width)/(GLfloat)(height), 0.1f, 500.0 );
}
```

如果loop等于3,我们就画最后一个视口(第四个视口)。它将位于窗口的左下部分。宽度和高度跟前几次设置一样。唯一跟第三个视口不同的是glViewport()的第一个参数为0.这告诉程序视口将在主窗口的左下部分。

第四个视口看起来如下:



透视投影视图设置同第二个视口。

```
if (loop==3) // 绘制右下角视口
{
    glViewport (0, 0, window_width/2, window_height/2);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
    gluPerspective( 45.0, (GLfloat)(width)/(GLfloat)(height), 0.1f, 500.0 );
}
```

下面的代码选择模型视图矩阵为当前矩阵，并重置它。然后清空深度缓存。我们在每个视口画之前清空深度缓存。注意到我们没有清除屏幕颜色，只是深度缓存！如果你没有清除深度缓存，你将看到物体的部分消失了，等等，很明显不美观！

```
glMatrixMode (GL_MODELVIEW);
glLoadIdentity ();

glClear (GL_DEPTH_BUFFER_BIT);
```

我们要画的第一副图为一个平坦的2维纹理方块。这个方块是在平行投影模式下画的，并且将会覆盖整个视口。因为我们用了平行投影投影模式，这儿没有第三维了，因此没必要在z轴进行变换。

记住我们第一个视口的左上角坐标维0,0,右下部分坐标为window\_width/2,window\_height/2.这意味着我们的四边形的右上坐标为window\_width/2,0,左上坐标为0,0,左下坐标为0,window\_height/2.右下坐标为window\_width/2,window\_height/2.请注意在平行投影投影模式下，我们能在象素级别上处理而不是单元级别(决定于我们的视口设置)

```
if (loop==0) // 绘制左上角的视图
{
    glBegin(GL_QUADS);
    glTexCoord2f(1.0f, 0.0f); glVertex2i(window_width/2, 0 );
    glTexCoord2f(0.0f, 0.0f); glVertex2i(0, 0 );
    glTexCoord2f(0.0f, 1.0f); glVertex2i(0, window_height/2);
    glTexCoord2f(1.0f, 1.0f); glVertex2i(window_width/2, window_height/2);
    glEnd();
```

}

第二个要画的图像是一个带光照的平滑球体。第二个视图是带透视的，因此我们首先必须做的是往屏幕里平移14个单位，然后在x, y, z轴旋转物体。

我们激活光照，画球体，然后关闭光照。这个球体半径为4个单元长度，围绕z轴的细分度为32,沿z轴的细分度也为32. 如果你还在犯迷糊，可以试着改变stacks或者slices的值为更小。通过减小stacks/slices的值，你就减少了球体的平滑度。

纹理坐标是自动产生的！

```
if (loop==1)                                // 绘制右上角的视图
{
    glTranslatef(0.0f,0.0f,-14.0f);

    glRotatef(xrot,1.0f,0.0f,0.0f);
    glRotatef(yrot,0.0f,1.0f,0.0f);
    glRotatef(zrot,0.0f,0.0f,1.0f);

    glEnable(GL_LIGHTING);
    gluSphere(quadric,4.0f,32,32);
    glDisable(GL_LIGHTING);
}
```

要画的第三幅图跟第一幅一样。但是是带透视的。它贴到屏幕有一定的角度并且有旋转。

我们把它往屏幕里移动2个单位。然后往后倾斜那个方块45度角。这让方块的顶部远离我们，而方块的底部则更靠近我们。

然后在z轴方向上旋转方块。画方块时，我们需要手工设置贴图坐标。

```
if (loop==2)                                // 绘制右下角的视图
{
    glTranslatef(0.0f,0.0f,-2.0f);
    glRotatef(-45.0f,1.0f,0.0f,0.0f);
    glRotatef(zrot/1.5f,0.0f,0.0f,1.0f);

    glBegin(GL_QUADS);
    glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f, 1.0f, 0.0f);
```

```

glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f, 1.0f, 0.0f);
glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, 0.0f);
glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, 0.0f);
glEnd();
}

```

如果我们在画第四副图，我们往屏幕里移动7个单位。然后把物体绕x,y,z轴旋转。我们激活光照给物体一些不错的阴影效果，然后在z轴上平移-2个单位。我们这样做的原因是让物体绕自己的中心旋转而不是绕某一端。这圆柱体两端宽1.5个单位。长度为4个单位并且绕轴上细分32个面片，沿轴细分16个面片。  
为了能绕中心旋转，我们需要平移柱体长度的一半，4的一半也即是2。  
在平移，旋转，然后再平移之后，我们画圆柱体，之后关闭光照。

```

if (loop==3)                                // 绘制左下角的视图
{
    glTranslatef(0.0f,0.0f,-7.0f);
    glRotatef(-xrot/2,1.0f,0.0f,0.0f);
    glRotatef(-yrot/2,0.0f,1.0f,0.0f);
    glRotatef(-zrot/2,0.0f,0.0f,1.0f);

    glEnable(GL_LIGHTING);
    glTranslatef(0.0f,0.0f,-2.0f);
    gluCylinder(quadric,1.5f,1.5f,4.0f,32,16);
    glDisable(GL_LIGHTING);
}
}

```

最后要做的事就是清空渲染管道。

```

glFlush ();
}

```

希望这个教程能解答所有你在做多视口中碰到的任何问题。代码并不难懂。它几乎跟标准的基本代码没什么区别。我们唯一真正修改的是视口设置是在画的主循环中。在所有视口画之前清空一次屏幕，然后清空各自深度缓存。

你可以用这些代码来在各自的视口中显示各种各样的图片，或在多视图中显示特定的物体。要做什么起决于你自己

我希望你们喜欢这个教程...如果你发现代码中的任何错误，或者你感觉你能让这个教程更好，请通知我(同样的，如果你看过我的翻译，发现有不当之处，请通知我)

### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第41课

第43课 >

## 第43课



Active WGL Bitmap Text With NeHe - 311.75

ACTIVE FreeType Text - 311.75

### 在OpenGL中使用FreeType库

使用FreeType库可以创建非常好看的文字，记住暴雪公司就是使用这个库的，就是那个做魔兽世界的。尝试一下吧，我只告诉你了基本的使用方式，你可以走的更远。

### 在OpenGL中使用FreeType库

这里是一个快速的介绍，它告诉你如何在OpenGL中使用FreeType渲染TrueType字体。使用这个库我们可以渲染反走样的文本，它看起来更加的漂亮。

#### 动机

这里我将给你两个例子，一个是用WGL的bitmap字体渲染得文字，另一个是用FreeType渲染得文字。

使用WGI渲染得文字是一些图像，当你放大它们时看起来如下：



foo

如果你使用GNU的FreeType库（暴雪公司也在它们的游戏中使用这个库），它将看起来更漂亮，如下所示，它具有了反走样：



## 创建程序

第一步你需要从下面的网站上下载FreeType库：<http://gnuwin32.sourceforge.net/packages/freetype.htm>

接着在你使用它创建一个新的程序时，你需要链接libfreetype.lib库，并包含FreeType的头文件。

现在我们已经能创建基于FreeType的程序了，但我们还不能运行它，因为我们需要freetype-6.dll文件。

好了，现在我们可以开始编写我们的程序了，我们从13课的代码开始，我们添加两个新的文件"freetype.cpp"和"freetype.h"。我们把和FreeType相关的内容放在这两个文件里。

好了，让我们从freetype.h开始吧。

按惯例我们包含一些需要的头文件

```
#ifndef FREE_NEHE_H
#define FREE_NEHE_H

//FreeType 头文件
#include <ft2build.h>
#include <freetype/freetype.h>
#include <freetype/ftglyph.h>
#include <freetype/ftoutln.h>
#include <freetype/fttrigon.h>

//OpenGL 头文件
#include <windows.h>
#include <GL/gl.h>
#include <GL/glu.h>

//STL 头文件
#include <vector>
#include <string>

//STL异常类
#include <stdexcept>
#pragma warning(disable: 4786)
```

我们将把每个字符需要的信息封装在一个结构中，这样就像使用WGL字体一样，我们可以分别控制每个字符的显示状态。

```
// 把所有的操作放在名字空间freetype中，这样可以避免与其他函数的冲突
namespace freetype
{

// 使用vector和string名字空间
using std::vector;
using std::string;

// 这个结构保存字体信息
struct font_data
{
    float h; // 字体的高度
    GLuint * textures; // 使用的纹理
}
```

```
GLuint list_base; // 显示列表的值
```

```
// 初始化结构
```

```
void init(const char * fname, unsigned int h);
```

```
// 清楚所有的资源
```

```
void clean();
```

```
};
```

最后一件事是定义我们输出字符串的原形:

```
// 把字符输出到屏幕
```

```
void print(const font_data &ft_font, float x, float y, const char *fmt, ...);
```

```
}
```

```
#endif
```

上面就是FreeType的头文件 , 下面我们看看怎样实现它

```
#include "freetype.h"
```

```
namespace freetype {
```

我们使用纹理去显示字符 , 在OpenGL中纹理大小必须为2的次方 , 这个函数用来字符的大小近似到这个值。所以我们有了如下的方程 :

```
// 这个函数返回比a大的 , 并且是最接近a的2的次方的数
```

```
inline int next_p2 (int a)
```

```
{
```

```

int rval=1;
// rval<<=1 Is A Prettier Way Of Writing rval*=2;
while(rval<a) rval<<=1;
return rval;
}

```

下面一个函数为make\_dlist, 它是这个代码的核心。它包含FT\_Face对象，它是FreeType用来保存字体信息的类，接着创建一个显示列表。

```

// 为给定的字符创建一个显示列表
void make_dlist ( FT_Face face, char ch, GLuint list_base, GLuint * tex_base ) {

// 载入给定字符的轮廓
if(FT_Load_Glyph( face, FT_Get_Char_Index( face, ch ), FT_LOAD_DEFAULT ))
throw std::runtime_error("FT_Load_Glyph failed");

// 保存轮廓对象
FT_Glyph glyph;
if(FT_Get_Glyph( face->glyph, &glyph ))
throw std::runtime_error("FT_Get_Glyph failed");

// 把轮廓转化为位图
FT_Glyph_To_Bitmap( &glyph, ft_render_mode_normal, 0, 1 );
FT_BitmapGlyph bitmap_glyph = (FT_BitmapGlyph)glyph;

// 保存位图
FT_Bitmap& bitmap=bitmap_glyph->bitmap;
}

}

```

现在我们已经从FreeType中获得了位图，我们需要把它转化为一个满足OpenGL纹理要求的位图。你必须知道，在OpenGL中位图表示黑白的数据，而在FreeType中我们使用8位的颜色表示位图，所以FreeType的位图可以保存亮度信息。

// 转化为OpenGL可以使用的大小

```
int width = next_p2( bitmap.width );
int height = next_p2( bitmap.rows );
```

// 保存位图数据

```
GLubyte* expanded_data = new GLubyte[ 2 * width * height];
```

// 这里我们使用8位表示亮度8位表示alpha值

```
for(int j=0; j < height; j++) {
    for(int i=0; i < width; i++) {
        expanded_data[2*(i+j*width)] = expanded_data[2*(i+j*width)+1] =
            (i>=bitmap.width || j>=bitmap.rows) ?
            0 : bitmap.buffer[i + bitmap.width*j];
    }
}
```

接下来我们选则字体纹理，并生成字体的贴图纹理

// 设置字体纹理的纹理过滤器

```
glBindTexture( GL_TEXTURE_2D, tex_base[ch]);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
```

// 邦定纹理

```
glTexImage2D( GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
    GL_LUMINANCE_ALPHA, GL_UNSIGNED_BYTE, expanded_data );
```

// 释放分配的内存

```
delete [] expanded_data;
```

接着创建一个显示列表，它用来绘制一个字符

// 创建显示列表

```
glNewList(list_base+ch,GL_COMPILE);
```

```
glBindTexture(GL_TEXTURE_2D,tex_base[ch]);
```

```
//首先我们向左移动一点
```

```
glTranslatef(bitmap_glyph->left,0,0);
```

```
//接着我们向下移动一点 , 这只对'g','y'之类的字符有用
```

```
//它使得所有的字符都有一个基线
```

```
glPushMatrix();
```

```
glTranslatef(0,bitmap_glyph->top-bitmap.rows,0);
```

```
// 计算位图中字符图像的宽度
```

```
float x=(float)bitmap.width / (float)width,
```

```
y=(float)bitmap.rows / (float)height;
```

```
//绘制一个正方形 , 显示字符
```

```
glBegin(GL_QUADS);
```

```
glTexCoord2d(0,0); glVertex2f(0,bitmap.rows);
```

```
glTexCoord2d(0,y); glVertex2f(0,0);
```

```
glTexCoord2d(x,y); glVertex2f(bitmap.width,0);
```

```
glTexCoord2d(x,0); glVertex2f(bitmap.width,bitmap.rows);
```

```
glEnd();
```

```
glPopMatrix();
```

```
glTranslatef(face->glyph->advance.x >> 6 ,0,0);
```

```
//结束显示列表的绘制
```

```
glEndList();
```

```
}
```

下面的函数将使用make\_dlist创建一个字符集的显示列表 , fname为你要使用的FreeType字符文件。

```
void font_data::init(const char * fname, unsigned int h) {
```

```
    // 保存纹理ID.
```

```
    textures = new GLuint[128];
```

```
this->h=h;
```

```
// 创建FreeType库
```

```
FT_Library library;
```

```
if (FT_Init_FreeType( &library ))
throw std::runtime_error("FT_Init_FreeType failed");

// 在FreeType库中保存字体信息的类叫做face
FT_Face face;

// 使用你输入的Freetype字符文件初始化face类
if (FT_New_Face( library, fname, 0, &face ))
throw std::runtime_error("FT_New_Face failed (there is probably a problem with your font file)");

// 在FreeType中使用1/64作为一个像素的高度所以我们需要缩放h来满足这个要求
FT_Set_Char_Size( face, h << 6, h << 6, 96, 96);

// 创建128个显示列表
list_base=glGenLists(128);
glGenTextures( 128, textures );
make_dlist(face,i,list_base,textures);

// 释放face类
FT_Done_Face(face);

// 释放FreeType库
FT_Done_FreeType(library);
}
```

下面的函数完成释放资源的工作

```
void font_data::clean() {
    glDeleteLists(list_base,128);
    glDeleteTextures(128,textures);
    delete [] textures;}
```

在print函数中要用到下面的两个方程，pushScreenCoordinateMatrix函数用来保存当前的矩阵，并设置视口与当前的窗口大小匹配。pop\_projection\_matrix函数用来返回pushScreenCoordinateMatrix保存的矩阵。reference manual.

// 保存当前的矩阵，并设置视口与当前的窗口大小匹配

```
inline void pushScreenCoordinateMatrix() {
    glPushAttrib(GL_TRANSFORM_BIT);
    GLint viewport[4];
    glGetIntegerv(GL_VIEWPORT, viewport);
    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluOrtho2D(viewport[0],viewport[2],viewport[1],viewport[3]);
    glPopAttrib();
}
```

//返回pushScreenCoordinateMatrix保存的矩阵

```
inline void pop_projection_matrix() {
    glPushAttrib(GL_TRANSFORM_BIT);
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();
    glPopAttrib();
```

我们的print函数和13课的函数非常的像，但在实现上有一些不同。我们实际上是使用2通道的纹理而不是图像。

// 输出文字

```
void print(const font_data &ft_font, float x, float y, const char *fmt, ...) {
    // 保存当前矩阵
    pushScreenCoordinateMatrix();

    GLuint font=ft_font.list_base;
    float h=ft_font.h/.63f;
    char text[256];
    va_list ap;
```

```
if (fmt == NULL)
*text=0;
else {
va_start(ap, fmt);
vsprintf(text, fmt, ap);
va_end(ap);
}

// 把输入的字符串按回车分割
const char *start_line=text;
vector<string> lines;
for(const char *c=text;*c;c++) {
if(*c=='\n') {
string line;
for(const char *n=start_line;n<c;n++) line.append(1,*n);
lines.push_back(line);
start_line=c+1;
}
}
if(start_line) {
string line;
for(const char *n=start_line;n<c;n++) line.append(1,*n);
lines.push_back(line);
}

glPushAttrib(GL_LIST_BIT | GL_CURRENT_BIT | GL_ENABLE_BIT | GL_TRANSFORM_BIT);
glMatrixMode(GL_MODELVIEW);
glDisable(GL_LIGHTING);
glEnable(GL_TEXTURE_2D);
glDisable(GL_DEPTH_TEST);
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

glListBase(font);

float modelview_matrix[16];
glGetFloatv(GL_MODELVIEW_MATRIX, modelview_matrix);

// 下面的代码完成具体的绘制过程
for(int i=0;i<lines.size();i++) {
glPushMatrix();
glLoadIdentity();
```

```
glTranslatef(x,y-h*i,0);
glMultMatrixf(modelview_matrix);

//调用显示列表绘制
glCallLists(lines[i].length(), GL_UNSIGNED_BYTE, lines[i].c_str());

glPopMatrix();
}

glPopAttrib();

pop_projection_matrix();
}

}
```

FreeType库我们就写好了，现我们在13课的代码上做一些修改，当然首先我们需要包含freetype.h的头文件

```
#include "freetype.h"
```

现在我们就可以调用freetype库绘制字符串了

```
// 保存我们创建的字体的信息
freetype::font_data our_font;
```

接下来使用test.ttf文件初始化字体

```
our_font.init("Test.ttf", 16);
```

在程序结束时记得释放内存资源

```
our_font.clean();
```

下面是我们具体的绘制函数

```
int DrawGLScene(GLvoid)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0f,0.0f,-1.0f);

    // 蓝色文字
    glColor3ub(0,0,0xff);

    // 绘制WGL文字
    glRasterPos2f(-0.40f, 0.35f);
    glPrint("Active WGL Bitmap Text With NeHe - %7.2f", cnt1);

    // 红色文字
    glColor3ub(0xff,0,0);

    glPushMatrix();
    glLoadIdentity();
    glRotatef(cnt1,0,0,1);
    glScalef(1.,8+.3*cos(cnt1/5),1);
    glTranslatef(-180,0,0);
    //绘制freetype文字
    freetype::print(our_font, 320, 200, "Active FreeType Text - %7.2f", cnt1);
    glPopMatrix();

    cnt1+=0.051f;
```

```
cnt2+=0.005f;  
return TRUE; // 成功返回  
}
```

最后我们介绍一些实用的创建字体的相关站点

OGLFT 非常漂亮的基于FreeType2的字体库，下面是它的站点<http://oglft.sourceforge.net>.

FTGL 是为OS X设计的第三方字体库. <http://homepages.paradise.net.nz/henryj/code/#FTGL>.

FNT 一个非FreeType库，它使用自己定义的字体格式，但它具有非常好的界面<http://plib.sourceforge.net/fnt>.

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

#### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

#### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢

积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第42课

第44课 >



## 第44课

### 3D 光晕

当镜头对准太阳的时候就会出现这种效果，模拟它非常的简单，一点数学和纹理贴图就够了。好好看看吧。

大家好,欢迎来到新的一课,在这一课中我们将扩展glCamera类,来实现镜头光晕的效果。在日常生活中,当我们对着光源看时,会发现强烈的反光。

为了完成这个效果,我们需要一些数学知识。首先,我们需要一些函数,用来检测某个点或球是否在当前的视景体内。接着我们需要一些纹理作为我们的光晕效果,我们可以把它贴在显示面上。

在我的上一个摄像机类里把下面函数写错了,现在修正如下:

```
void glCamera::SetPerspective()
{
    GLfloat Matrix[16];
    glVector v;

// 根据当前的偏转角旋转视线
glRotatef(m_HeadingDegrees, 0.0f, 1.0f, 0.0f);
glRotatef(m_PitchDegrees, 1.0f, 0.0f, 0.0f);

// 返回模型变换矩阵
glGetFloatv(GL_MODELVIEW_MATRIX, Matrix);
```

```
// 获得视线的方向  
m_DirectionVector.i = Matrix[8];  
m_DirectionVector.j = Matrix[9];  
m_DirectionVector.k = -Matrix[10];  
  
// 重置矩阵  
glLoadIdentity();  
  
// 旋转场景  
glRotatef(m_PitchDegrees, 1.0f, 0.0f, 0.0f);  
glRotatef(m_HeadingDegrees, 0.0f, 1.0f, 0.0f);  
  
// 设置当前摄像机的位置  
v = m_DirectionVector;  
v *= m_ForwardVelocity;  
m_Position.x += v.i;  
m_Position.y += v.j;  
m_Position.z += v.k;  
  
// 变换到新的位置  
glTranslatef(-m_Position.x, -m_Position.y, -m_Position.z);  
}
```

好了，我们现在开始吧。我将使用4个对立的纹理来制造我们的镜头光晕，第一和二个光晕图像被放置在光源处，第三和第四个图像将根据视点的位置和方向动态的生成。纹理的图像如下所示：

Big Glow



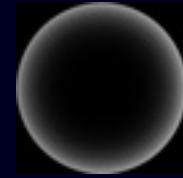
Streaks



Glow



Halo



现在你在头脑里应该有了一个大概地图像了吧。我们来说说何时我们应该绘制光晕，一般来说平时我们是看不见这些光晕的，只有当我们对准光源的时候才能看见这些。所以我们首先要获得视景体的数据，下面的函数可以帮我们完成这个功能。

```
// 获得当前视景体的6个平面方程的参数
void glCamera::UpdateFrustum()
{
    GLfloat clip[16];
    GLfloat proj[16];
    GLfloat modl[16];
    GLfloat t;

//返回投影矩阵
glGetFloatv( GL_PROJECTION_MATRIX, proj );

//返回模型变换矩阵
glGetFloatv( GL_MODELVIEW_MATRIX, modl );

//计算剪切矩阵，即上面两个矩阵的乘积
clip[ 0 ] = modl[ 0 ] * proj[ 0 ] + modl[ 1 ] * proj[ 4 ] + modl[ 2 ] * proj[ 8 ] + modl[ 3 ] * proj[ 12 ];
clip[ 1 ] = modl[ 0 ] * proj[ 1 ] + modl[ 1 ] * proj[ 5 ] + modl[ 2 ] * proj[ 9 ] + modl[ 3 ] * proj[ 13 ];
clip[ 2 ] = modl[ 0 ] * proj[ 2 ] + modl[ 1 ] * proj[ 6 ] + modl[ 2 ] * proj[ 10 ] + modl[ 3 ] * proj[ 14 ];
clip[ 3 ] = modl[ 0 ] * proj[ 3 ] + modl[ 1 ] * proj[ 7 ] + modl[ 2 ] * proj[ 11 ] + modl[ 3 ] * proj[ 15 ];

clip[ 4 ] = modl[ 4 ] * proj[ 0 ] + modl[ 5 ] * proj[ 4 ] + modl[ 6 ] * proj[ 8 ] + modl[ 7 ] * proj[ 12 ];
clip[ 5 ] = modl[ 4 ] * proj[ 1 ] + modl[ 5 ] * proj[ 5 ] + modl[ 6 ] * proj[ 9 ] + modl[ 7 ] * proj[ 13 ];
clip[ 6 ] = modl[ 4 ] * proj[ 2 ] + modl[ 5 ] * proj[ 6 ] + modl[ 6 ] * proj[ 10 ] + modl[ 7 ] * proj[ 14 ];
clip[ 7 ] = modl[ 4 ] * proj[ 3 ] + modl[ 5 ] * proj[ 7 ] + modl[ 6 ] * proj[ 11 ] + modl[ 7 ] * proj[ 15 ];
```

```

clip[ 8] = modl[ 8] * proj[ 0] + modl[ 9] * proj[ 4] + modl[10] * proj[ 8] + modl[11] * proj[12];
clip[ 9] = modl[ 8] * proj[ 1] + modl[ 9] * proj[ 5] + modl[10] * proj[ 9] + modl[11] * proj[13];
clip[10] = modl[ 8] * proj[ 2] + modl[ 9] * proj[ 6] + modl[10] * proj[10] + modl[11] * proj[14];
clip[11] = modl[ 8] * proj[ 3] + modl[ 9] * proj[ 7] + modl[10] * proj[11] + modl[11] * proj[15];

```

```

clip[12] = modl[12] * proj[ 0] + modl[13] * proj[ 4] + modl[14] * proj[ 8] + modl[15] * proj[12];
clip[13] = modl[12] * proj[ 1] + modl[13] * proj[ 5] + modl[14] * proj[ 9] + modl[15] * proj[13];
clip[14] = modl[12] * proj[ 2] + modl[13] * proj[ 6] + modl[14] * proj[10] + modl[15] * proj[14];
clip[15] = modl[12] * proj[ 3] + modl[13] * proj[ 7] + modl[14] * proj[11] + modl[15] * proj[15];

```

//提取右面的平面方程系数

```

m_Frustum[0][0] = clip[ 3] - clip[ 0];
m_Frustum[0][1] = clip[ 7] - clip[ 4];
m_Frustum[0][2] = clip[11] - clip[ 8];
m_Frustum[0][3] = clip[15] - clip[12];
t = GLfloat(sqrt( m_Frustum[0][0] * m_Frustum[0][0] + m_Frustum[0][1] * m_Frustum[0][1] +
m_Frustum[0][2] * m_Frustum[0][2] ));
m_Frustum[0][0] /= t;
m_Frustum[0][1] /= t;
m_Frustum[0][2] /= t;
m_Frustum[0][3] /= t;

```

//提取左面的平面方程系数

```

m_Frustum[1][0] = clip[ 3] + clip[ 0];
m_Frustum[1][1] = clip[ 7] + clip[ 4];
m_Frustum[1][2] = clip[11] + clip[ 8];
m_Frustum[1][3] = clip[15] + clip[12];
t = GLfloat(sqrt( m_Frustum[1][0] * m_Frustum[1][0] + m_Frustum[1][1] * m_Frustum[1][1] +
m_Frustum[1][2] * m_Frustum[1][2] ));
m_Frustum[1][0] /= t;
m_Frustum[1][1] /= t;
m_Frustum[1][2] /= t;
m_Frustum[1][3] /= t;

```

//提取下面的平面方程系数

```

m_Frustum[2][0] = clip[ 3] + clip[ 1];
m_Frustum[2][1] = clip[ 7] + clip[ 5];
m_Frustum[2][2] = clip[11] + clip[ 9];
m_Frustum[2][3] = clip[15] + clip[13];
t = GLfloat(sqrt( m_Frustum[2][0] * m_Frustum[2][0] + m_Frustum[2][1] * m_Frustum[2][1] +
m_Frustum[2][2] * m_Frustum[2][2] ));
m_Frustum[2][0] /= t;

```

```
m_Frustum[2][1] /= t;  
m_Frustum[2][2] /= t;  
m_Frustum[2][3] /= t;
```

//提取上面的平面方程系数

```
m_Frustum[3][0] = clip[ 3] - clip[ 1];  
m_Frustum[3][1] = clip[ 7] - clip[ 5];  
m_Frustum[3][2] = clip[11] - clip[ 9];  
m_Frustum[3][3] = clip[15] - clip[13];  
t = GLfloat(sqrt( m_Frustum[3][0] * m_Frustum[3][0] + m_Frustum[3][1] * m_Frustum[3][1] +  
m_Frustum[3][2] * m_Frustum[3][2] ));  
m_Frustum[3][0] /= t;  
m_Frustum[3][1] /= t;  
m_Frustum[3][2] /= t;  
m_Frustum[3][3] /= t;
```

//提取远面的平面方程系数

```
m_Frustum[4][0] = clip[ 3] - clip[ 2];  
m_Frustum[4][1] = clip[ 7] - clip[ 6];  
m_Frustum[4][2] = clip[11] - clip[10];  
m_Frustum[4][3] = clip[15] - clip[14];  
t = GLfloat(sqrt( m_Frustum[4][0] * m_Frustum[4][0] + m_Frustum[4][1] * m_Frustum[4][1] +  
m_Frustum[4][2] * m_Frustum[4][2] ));  
m_Frustum[4][0] /= t;  
m_Frustum[4][1] /= t;  
m_Frustum[4][2] /= t;  
m_Frustum[4][3] /= t;
```

//提取近面的平面方程系数

```
m_Frustum[5][0] = clip[ 3] + clip[ 2];  
m_Frustum[5][1] = clip[ 7] + clip[ 6];  
m_Frustum[5][2] = clip[11] + clip[10];  
m_Frustum[5][3] = clip[15] + clip[14];  
t = GLfloat(sqrt( m_Frustum[5][0] * m_Frustum[5][0] + m_Frustum[5][1] * m_Frustum[5][1] +  
m_Frustum[5][2] * m_Frustum[5][2] ));  
m_Frustum[5][0] /= t;  
m_Frustum[5][1] /= t;  
m_Frustum[5][2] /= t;  
m_Frustum[5][3] /= t;  
}
```

现在我们可以测试一个点或圆是否在视景体内了。下面的函数可以测试一个点是否在视景体内。

```
BOOL glCamera::PointInFrustum(glPoint p)
{
    int i;
    for(i = 0; i < 6; i++)
    {
        if(m_Frustum[i][0] * p.x + m_Frustum[i][1] * p.y + m_Frustum[i][2] * p.z + m_Frustum[i][3] <= 0)
        {
            return(FALSE);
        }
    }
    return(TRUE);
}
```

下面的函数用来测试某个点是否位于当前场景物体的前面:

```
bool glCamera::IsOccluded(glPoint p)
{
    GLint viewport[4];
    GLdouble mvmatrix[16], projmatrix[16];
    GLdouble winx, winy, winz;
    GLdouble flareZ;
    GLfloat bufferZ;

    glGetIntegerv (GL_VIEWPORT, viewport);
    glGetDoublev (GL_MODELVIEW_MATRIX, mvmatrix);
    glGetDoublev (GL_PROJECTION_MATRIX, projmatrix);

    // 返回顶点p在单位立方体中的位置
    gluProject(p.x, p.y, p.z, mvmatrix, projmatrix, viewport, &winx, &winy, &winz);
    flareZ = winz;

    // 读取点(winx,winy)的深度坐标
```

```
glReadPixels(winx, winy, 1, 1, GL_DEPTH_COMPONENT, GL_FLOAT, &bufferZ);

// 如果深度坐标小于点的坐标，则返回true
if (bufferZ < flareZ)
    return true;
//否则返回false
else
    return false;
}
```

我们通过检测光源是否正对我们的视线来决定是否绘制光晕，但如果你的视点超过了光源的位置，则会发生看不见光晕的现象。为了避免这种现象，我们在移动视点的使用，也相应的移动我们的光源。为了在视点和光源之间绘制多个光晕，我们需要计算之间的向量，下面的代码完成这个功能：

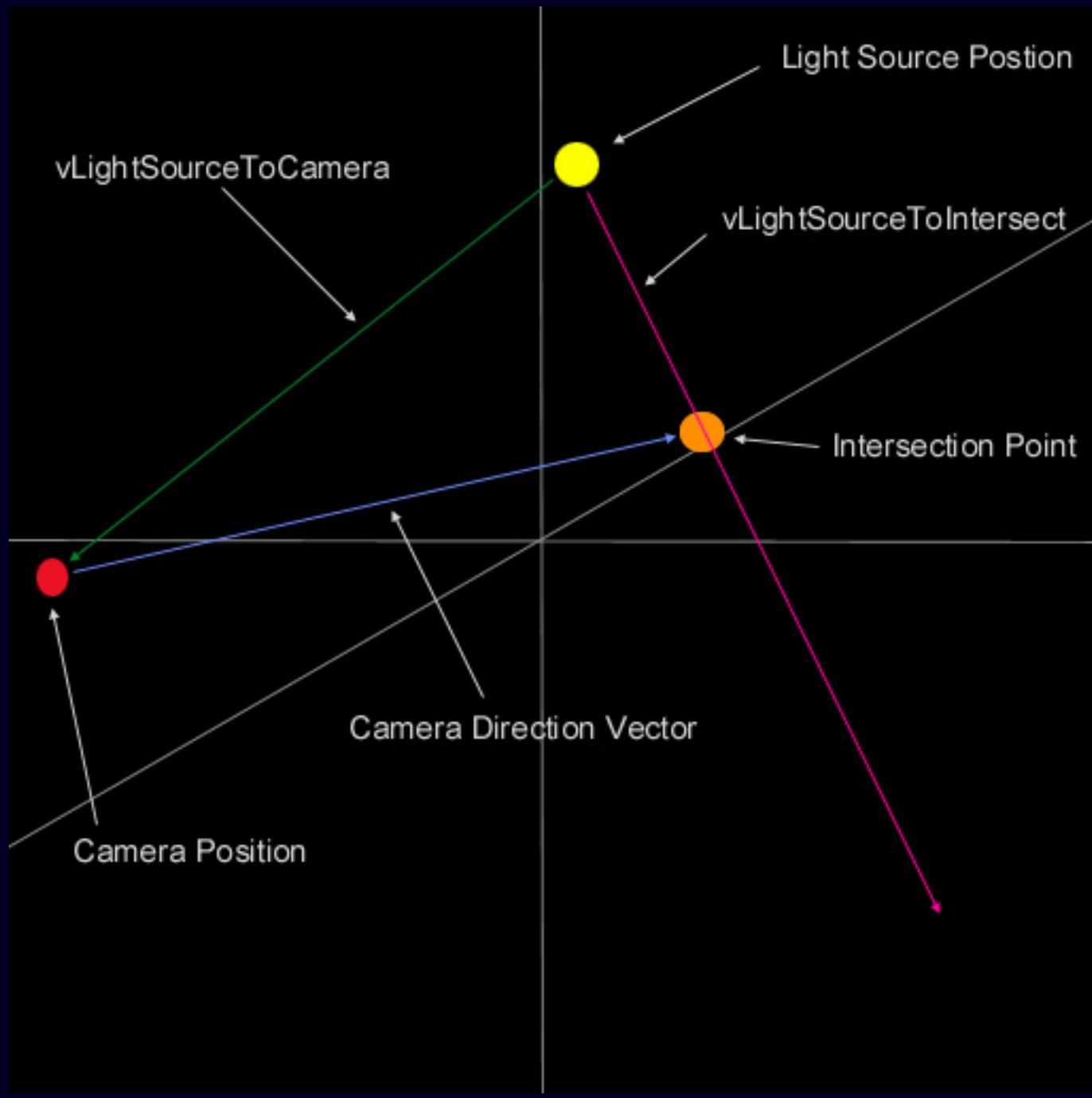
```
//下面的函数完成具体的渲染光晕的任务
void glCamera::RenderLensFlare()
{
    GLfloat Length = 0.0f;

// 如果我们的光源在我们的视线范围内，则绘制它
if(SphereInFrustum(m_LightSourcePos, 1.0f) == TRUE)
{
    vLightSourceToCamera = m_Position - m_LightSourcePos; // 计算光源到我们视线的距离
    Length = vLightSourceToCamera.Magnitude();

//下面三个函数计算光源位置到光晕结束位置之间的向量
    ptIntersect = m_DirectionVector * Length;
    ptIntersect += m_Position;
    vLightSourceToIntersect = ptIntersect - m_LightSourcePos;
    Length = vLightSourceToIntersect.Magnitude();
    vLightSourceToIntersect.Normalize();

    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE);
    glDisable(GL_DEPTH_TEST);
    glEnable(GL_TEXTURE_2D);
```

首先我们需要找到光源位置和视点位置之间的向量，接下来我们需要在视线的方向设置一个插值点，这个点的距离必须和光源位置和视点位置之间的距离相等。完成以后，我们找出可以产生光晕的方向，即下图红线的方向，在这个线上我们可以绘制我们的光晕。



```
if (!IsOccluded(m_LightSourcePos)) //如果光晕可见
```

```
{  
    // 渲染中间的光晕  
    RenderBigGlow(0.60f, 0.60f, 0.8f, 1.0f, m_LightSourcePos, 16.0f);  
    RenderStreaks(0.60f, 0.60f, 0.8f, 1.0f, m_LightSourcePos, 16.0f);  
    RenderGlow(0.8f, 0.8f, 1.0f, 0.5f, m_LightSourcePos, 3.5f);  
  
    //绘制到光晕结束位置的0.1处的光晕  
    pt = vLightSourceToIntersect * (Length * 0.1f);  
    pt += m_LightSourcePos;  
    RenderGlow(0.9f, 0.6f, 0.4f, 0.5f, pt, 0.6f);  
  
    //绘制到光晕结束位置的0.15处的光晕  
    pt = vLightSourceToIntersect * (Length * 0.15f);  
    pt += m_LightSourcePos;  
    RenderHalo(0.8f, 0.5f, 0.6f, 0.5f, pt, 1.7f);  
  
    //绘制到光晕结束位置的0.175处的光晕  
    pt = vLightSourceToIntersect * (Length * 0.175f);  
    pt += m_LightSourcePos;  
    RenderHalo(0.9f, 0.2f, 0.1f, 0.5f, pt, 0.83f);  
  
    //绘制到光晕结束位置的0.285处的光晕  
    pt = vLightSourceToIntersect * (Length * 0.285f);  
    pt += m_LightSourcePos;  
    RenderHalo(0.7f, 0.7f, 0.4f, 0.5f, pt, 1.6f);  
  
    //绘制到光晕结束位置的0.2755处的光晕  
    pt = vLightSourceToIntersect * (Length * 0.2755f);  
    pt += m_LightSourcePos;  
    RenderGlow(0.9f, 0.9f, 0.2f, 0.5f, pt, 0.8f);  
  
    //绘制到光晕结束位置的0.4775处的光晕  
    pt = vLightSourceToIntersect * (Length * 0.4775f);  
    pt += m_LightSourcePos;  
    RenderGlow(0.93f, 0.82f, 0.73f, 0.5f, pt, 1.0f);  
  
    //绘制到光晕结束位置的0.49处的光晕  
    pt = vLightSourceToIntersect * (Length * 0.49f);  
    pt += m_LightSourcePos;  
    RenderHalo(0.7f, 0.6f, 0.5f, 0.5f, pt, 1.4f);  
  
    //绘制到光晕结束位置的0.65处的光晕  
    pt = vLightSourceToIntersect * (Length * 0.65f);
```

```

pt += m_LightSourcePos;
RenderGlow(0.7f, 0.8f, 0.3f, 0.5f, pt, 1.8f);

//绘制到光晕结束位置的0.63处的光晕
pt = vLightSourceToIntersect * (Length * 0.63f);
pt += m_LightSourcePos;
RenderGlow(0.4f, 0.3f, 0.2f, 0.5f, pt, 1.4f);

//绘制到光晕结束位置的0.8处的光晕
pt = vLightSourceToIntersect * (Length * 0.8f);
pt += m_LightSourcePos;
RenderHalo(0.7f, 0.5f, 0.5f, 0.5f, pt, 1.4f);

//绘制到光晕结束位置的0.7825处的光晕
pt = vLightSourceToIntersect * (Length * 0.7825f);
pt += m_LightSourcePos;
RenderGlow(0.8f, 0.5f, 0.1f, 0.5f, pt, 0.6f);

//绘制到光晕结束位置的1.0处的光晕
pt = vLightSourceToIntersect * (Length * 1.0f);
pt += m_LightSourcePos;
RenderHalo(0.5f, 0.5f, 0.7f, 0.5f, pt, 1.7f);

//绘制到光晕结束位置的0.975处的光晕
pt = vLightSourceToIntersect * (Length * 0.975f);
pt += m_LightSourcePos;
RenderGlow(0.4f, 0.1f, 0.9f, 0.5f, pt, 2.0f);

}

glDisable(GL_BLEND );
glEnable(GL_DEPTH_TEST);
glDisable(GL_TEXTURE_2D);
}
}

```

好了，下面的函数用来绘制四种不同的光晕

```

//绘制Halo形的光晕
void glCamera::RenderHalo(GLfloat r, GLfloat g, GLfloat b, GLfloat a, glPoint p, GLfloat scale)

```

```
{  
    glPoint q[4];  
  
    q[0].x = (p.x - scale);  
    q[0].y = (p.y - scale);  
  
    q[1].x = (p.x - scale);  
    q[1].y = (p.y + scale);  
  
    q[2].x = (p.x + scale);  
    q[2].y = (p.y - scale);  
  
    q[3].x = (p.x + scale);  
    q[3].y = (p.y + scale);  
  
    glPushMatrix();  
    glTranslatef(p.x, p.y, p.z);  
    glRotatef(-m_HeadingDegrees, 0.0f, 1.0f, 0.0f);  
    glRotatef(-m_PitchDegrees, 1.0f, 0.0f, 0.0f);  
    glBindTexture(GL_TEXTURE_2D, m_HaloTexture);  
    glColor4f(r, g, b, a);  
  
    glBegin(GL_TRIANGLE_STRIP);  
        glTexCoord2f(0.0f, 0.0f);  
        glVertex2f(q[0].x, q[0].y);  
        glTexCoord2f(0.0f, 1.0f);  
        glVertex2f(q[1].x, q[1].y);  
        glTexCoord2f(1.0f, 0.0f);  
        glVertex2f(q[2].x, q[2].y);  
        glTexCoord2f(1.0f, 1.0f);  
        glVertex2f(q[3].x, q[3].y);  
    glEnd();  
    glPopMatrix();  
}
```

### //绘制Glow形的光晕

```
void glCamera::RenderGlow(GLfloat r, GLfloat g, GLfloat b, GLfloat a, glPoint p, GLfloat scale)  
{  
    glPoint q[4];  
  
    q[0].x = (p.x - scale);  
    q[0].y = (p.y - scale);  
  
    q[1].x = (p.x - scale);
```

```
q[1].y = (p.y + scale);
q[2].x = (p.x + scale);
q[2].y = (p.y - scale);

q[3].x = (p.x + scale);
q[3].y = (p.y + scale);

glPushMatrix();
glTranslatef(p.x, p.y, p.z);
glRotatef(-m_HeadingDegrees, 0.0f, 1.0f, 0.0f);
glRotatef(-m_PitchDegrees, 1.0f, 0.0f, 0.0f);
glBindTexture(GL_TEXTURE_2D, m_GlowTexture);
glColor4f(r, g, b, a);

glBegin(GL_TRIANGLE_STRIP);
glTexCoord2f(0.0f, 0.0f);
 glVertex2f(q[0].x, q[0].y);
glTexCoord2f(0.0f, 1.0f);
 glVertex2f(q[1].x, q[1].y);
glTexCoord2f(1.0f, 0.0f);
 glVertex2f(q[2].x, q[2].y);
glTexCoord2f(1.0f, 1.0f);
 glVertex2f(q[3].x, q[3].y);
glEnd();
glPopMatrix();
}
```

//绘制BigGlow形的光晕

```
void glCamera::RenderBigGlow(GLfloat r, GLfloat g, GLfloat b, GLfloat a, glPoint p, GLfloat scale)
{
glPoint q[4];

q[0].x = (p.x - scale);
q[0].y = (p.y - scale);

q[1].x = (p.x - scale);
q[1].y = (p.y + scale);

q[2].x = (p.x + scale);
q[2].y = (p.y - scale);

q[3].x = (p.x + scale);
q[3].y = (p.y + scale);
```

```
glPushMatrix();
glTranslatef(p.x, p.y, p.z);
glRotatef(-m_HeadingDegrees, 0.0f, 1.0f, 0.0f);
glRotatef(-m_PitchDegrees, 1.0f, 0.0f, 0.0f);
glBindTexture(GL_TEXTURE_2D, m_BigGlowTexture);
glColor4f(r, g, b, a);

glBegin(GL_TRIANGLE_STRIP);
glTexCoord2f(0.0f, 0.0f);
 glVertex2f(q[0].x, q[0].y);
glTexCoord2f(0.0f, 1.0f);
 glVertex2f(q[1].x, q[1].y);
glTexCoord2f(1.0f, 0.0f);
 glVertex2f(q[2].x, q[2].y);
glTexCoord2f(1.0f, 1.0f);
 glVertex2f(q[3].x, q[3].y);
glEnd();
glPopMatrix();
}
```

//绘制Streaks形的光晕

```
void glCamera::RenderStreaks(GLfloat r, GLfloat g, GLfloat b, GLfloat a, glPoint p, GLfloat scale)
{
glPoint q[4];

q[0].x = (p.x - scale);
q[0].y = (p.y - scale);

q[1].x = (p.x - scale);
q[1].y = (p.y + scale);

q[2].x = (p.x + scale);
q[2].y = (p.y - scale);

q[3].x = (p.x + scale);
q[3].y = (p.y + scale);

glPushMatrix();
glTranslatef(p.x, p.y, p.z);
glRotatef(-m_HeadingDegrees, 0.0f, 1.0f, 0.0f);
glRotatef(-m_PitchDegrees, 1.0f, 0.0f, 0.0f);
glBindTexture(GL_TEXTURE_2D, m_StreakTexture);
glColor4f(r, g, b, a);
```

```
glBegin(GL_TRIANGLE_STRIP);
glTexCoord2f(0.0f, 0.0f);
glVertex2f(q[0].x, q[0].y);
glTexCoord2f(0.0f, 1.0f);
glVertex2f(q[1].x, q[1].y);
glTexCoord2f(1.0f, 0.0f);
glVertex2f(q[2].x, q[2].y);
glTexCoord2f(1.0f, 1.0f);
glVertex2f(q[3].x, q[3].y);
glEnd();
glPopMatrix();
}
```

你可以使用w,s,a,d变换摄像机的方向，1，2显示/关闭各种信息参数。C给摄像机一个固定的速度，X停止它。

上面就是这个教程的全部了，所有的问题，评论和抱怨都欢迎。当然我不是第一个作这个效果的人，下面是其他方面相关的文章：

<http://www.gamedev.net/reference/articles/article874.asp>

<http://www.gamedev.net/reference/articles/article813.asp>

<http://www.opengl.org/developers/code/mjktips/lensflare/>

<http://www.markmorley.com/opengl/frustumculling.html>

[http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion\\_test.txt](http://oss.sgi.com/projects/ogl-sample/registry/HP/occlusion_test.txt)

#### 版权与使用声明：

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

#### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。



### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

&lt; 第43课

第45课 &gt;

## 第45课



### 顶点缓存

你想更快地绘制么？直接操作显卡吧，这可是当前的图形技术，不要犹豫，我带你入门。接下来，你自己向前走吧。

速度是3D程序中最重要的指标，你必须限制绘制的多边形的个数，或者提高显卡绘制多边形的效率。显卡最近增加了一个新的扩展，叫做顶点缓存VS，它直接把顶点放置在显卡中的高速缓存中，极大的增加了绘制速度。

在这个教程里，我们会加载一个高度图，使用顶点数组高效的把网格数据发送到OpenGL里，并使用VBO扩展把顶点数据放入高效的显存里。

现在让我们开始吧，我们先来定义一些程序参数。

```
#define MESH_RESOLUTION 4.0f           // 每个顶点使用的像素  
#define MESH_HEIGHTSCALE 1.0f          // 高度的缩放比例  
//#define NO_VBOS                  // 如果定义将不使用VBO扩展
```

```
// 定义VBO扩展它们在glext.h头文件中被定义  
#define GL_ARRAY_BUFFER_ARB 0x8892  
#define GL_STATIC_DRAW_ARB 0x88E4
```

```
typedef void (APIENTRY * PFNGLBINDBUFFERARBPROC) (GLenum target, GLuint buffer);
typedef void (APIENTRY * PFNGLDELETEBUFFERSARBPROC) (GLsizei n, const GLuint *buffers);
typedef void (APIENTRY * PFNGLGENBUFFERSARBPROC) (GLsizei n, GLuint *buffers);
typedef void (APIENTRY * PFNGLBUFFERDATAARBPROC) (GLenum target, int size, const GLvoid
*data, GLenum usage);
```

## // VBO 扩展函数的指针

```
PFNGLGENBUFFERSARBPROC glGenBuffersARB = NULL; // 创建缓存名称
PFNGLBINDBUFFERARBPROC glBindBufferARB = NULL; // 绑定缓存
PFNGLBUFFERDATAARBPROC glBufferDataARB = NULL; // 绑定缓存数据
PFNGLDELETEBUFFERSARBPROC glDeleteBuffersARB = NULL; // 删除缓存
```

现在我们来定义自己的网格类：

```
class CVert // 顶点类
{
public:
    float x;
    float y;
    float z;
};

typedef CVert CVec;

class CTexCoord // 纹理坐标类
{
public:
    float u;
    float v;
};

//网格类
class CMesh
{
public:
    // 网格数据
    int             m_nVertexCount;           // 顶点个数
    CVert*          m_pVertices;              // 顶点数据的指针
    CTexCoord*      m_pTexCoords;             // 顶点的纹理坐标
};
```

```

unsigned int m_nTextureId; // 纹理的ID

unsigned int m_nVBOVertices; // 顶点缓存对象的名称
unsigned int m_nVBOTexCoords; // 顶点纹理缓存对象的名称

AUX_RGBImageRec* m_pTextureImage; // 高度数据

public:
CMesh(); // 构造函数
~CMesh(); // 析构函数

// 载入高度图
bool LoadHeightmap( char* szPath, float flHeightScale, float flResolution );
// 返回单个点的高度
float PtHeight( int nX, int nY );
// 创建顶点缓存对象
void BuildVBOs();
};

}

```

大部分代码都很简单，这里不多加解释。

下面我们来定义一些全局变量：

```

bool g_fVBOSupported = false; // 是否支持顶点缓存对象
CMesh* g_pMesh = NULL; // 网格数据
float g_fYRot = 0.0f; // 旋转角度
int g_nFPS = 0, g_nFrames = 0; // 帧率计数器
DWORD g_dwLastFPS = 0; // 上一帧的计数

```

下面的代码加载高度图,它和34课的内容差不多,在这里不多加解释了:

```

//加载高度图
bool CMesh :: LoadHeightmap( char* szPath, float flHeightScale, float flResolution )
{

```

```
FILE* fTest = fopen( szPath, "r" );
if( !fTest )
    return false;
fclose( fTest );

// 加载图像文件
m_pTextureImage = auxDIBImageLoad( szPath );

// 读取顶点数据
m_nVertexCount = (int) ( m_pTextureImage->sizeX * m_pTextureImage->sizeY * 6 / ( flResolution *
flResolution ) );
m_pVertices = new CVec[m_nVertexCount];
m_pTexCoords = new CTexCoord[m_nVertexCount];
int nX, nZ, nTri, nIndex=0;
float flX, flZ;
for( nZ = 0; nZ < m_pTextureImage->sizeY; nZ += (int) flResolution )
{
    for( nX = 0; nX < m_pTextureImage->sizeX; nX += (int) flResolution )
    {
        for( nTri = 0; nTri < 6; nTri++ )
        {
            flX = (float) nX + ( ( nTri == 1 || nTri == 2 || nTri == 5 ) ? flResolution : 0.0f );
            flZ = (float) nZ + ( ( nTri == 2 || nTri == 4 || nTri == 5 ) ? flResolution : 0.0f );

            m_pVertices[nIndex].x = flX - ( m_pTextureImage->sizeX / 2 );
            m_pVertices[nIndex].y = PtHeight( (int) flX, (int) flZ ) * flHeightScale;
            m_pVertices[nIndex].z = flZ - ( m_pTextureImage->sizeY / 2 );

            m_pTexCoords[nIndex].u = flX / m_pTextureImage->sizeX;
            m_pTexCoords[nIndex].v = flZ / m_pTextureImage->sizeY;

            nIndex++;
        }
    }
}

// 载入纹理，它和高度图是同一副图像
glGenTextures( 1, &m_nTextureId );
 glBindTexture( GL_TEXTURE_2D, m_nTextureId );
 glTexImage2D( GL_TEXTURE_2D, 0, 3, m_pTextureImage->sizeX, m_pTextureImage->sizeY, 0,
GL_RGB, GL_UNSIGNED_BYTE, m_pTextureImage->data );
 glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);
```

```

glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);

// 释放纹理数据
if( m_pTextureImage )
{
    if( m_pTextureImage->data )
        free( m_pTextureImage->data );
    free( m_pTextureImage );
}
return true;
}

```

下面的代码用来计算(x,y)处的亮度

```

//计算(x,y)处的亮度
float CMesh :: PtHeight( int nX, int nY )
{
    int nPos = ( ( nX % m_pTextureImage->sizeX ) + ( ( nY % m_pTextureImage->sizeY ) *
m_pTextureImage->sizeX ) ) * 3;
    float fIR = (float) m_pTextureImage->data[ nPos ];           // 返回红色分量
    float fIG = (float) m_pTextureImage->data[ nPos + 1 ];       // 返回绿色分量
    float fIB = (float) m_pTextureImage->data[ nPos + 2 ];       // 返回蓝色分量
    return ( 0.299f * fIR + 0.587f * fIG + 0.114f * fIB );     // 计算亮度
}

```

下面的代码把顶点数据绑定到顶点缓存，即把内存中的数据发送到显存

```

void CMesh :: BuildVBOs()
{
    glGenBuffersARB( 1, &m_nVBOVertices );                                // 创建一个顶点缓存，并把顶
点数据绑定到缓存
    glBindBufferARB( GL_ARRAY_BUFFER_ARB, m_nVBOVertices );
    glBufferDataARB( GL_ARRAY_BUFFER_ARB, m_nVertexCount*3*sizeof(float), m_pVertices,
GL_STATIC_DRAW_ARB );
}

```

```

glGenBuffersARB( 1, &m_nVBOTexCoords ); // 创建一个纹理缓存，并把纹理数据绑定到缓存
glBindBufferARB( GL_ARRAY_BUFFER_ARB, m_nVBOTexCoords );
glBufferDataARB( GL_ARRAY_BUFFER_ARB, m_nVertexCount*2*sizeof(float), m_pTexCoords,
GL_STATIC_DRAW_ARB );

// 删除分配的内存
delete [] m_pVertices; m_pVertices = NULL;
delete [] m_pTexCoords; m_pTexCoords = NULL

}

```

好了，现在到了初始化的地方了。首先我将分配并载入纹理数据。接着检测是否支持VBO扩展。如果支持我们将把函数指针和它对应的函数关联起来，如果不支持将只返回数据。

```

//初始化
BOOL Initialize (GL_Window* window, Keys* keys)
{
    g_window      = window;
    g_keys        = keys;

    // 载入纹理数据
    g_pMesh = new CMesh();
    if( !g_pMesh->LoadHeightmap( "terrain.bmp",
                                MESH_HEIGHTSCALE,
                                MESH_RESOLUTION ) )
    {
        MessageBox( NULL, "Error Loading Heightmap", "Error", MB_OK );
        return false;
    }

    // 检测是否支持VBO扩展
#ifndef NO_VBOS
    g_fVBOSupported = IsExtensionSupported( "GL_ARB_vertex_buffer_object" );
    if( g_fVBOSupported )
    {
        // 获得函数的指针
        glGenBuffersARB = (PFNGLGENBUFFERSARBPROC) wglGetProcAddress("glGenBuffersARB");
        glBindBufferARB = (PFNGLBINDBUFFERARBPROC) wglGetProcAddress("glBindBufferARB");

```

```

glBufferDataARB = (PFNGLBUFFERDATAARBPROC) wglGetProcAddress("glBufferDataARB");
glDeleteBuffersARB = (PFNGLDELETEBUFFERSARBPROC) wglGetProcAddress
("glDeleteBuffersARB");
// 创建VBO对象
g_pMesh->BuildVBOs();
}
#else
g_fVBOSupported = false;
#endif
//设置OpenGL状态
glClearColor (0.0f, 0.0f, 0.0f, 0.5f);
glClearDepth (1.0f);
glDepthFunc (GL_LEQUAL);
glEnable (GL_DEPTH_TEST);
glShadeModel (GL_SMOOTH);
glHint (GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);
glEnable(GL_TEXTURE_2D);
glColor4f( 1.0f, 1.0f, 1.0f, 1.0f );

return TRUE;
}

```

下面的函数用来检测是否包含特定的扩展名称

```

// 返回是否支持指定的扩展
bool IsExtensionSupported( char* szTargetExtension )
{
    const unsigned char *pszExtensions = NULL;
    const unsigned char *pszStart;
    unsigned char *pszWhere, *pszTerminator;

    pszWhere = (unsigned char *) strchr( szTargetExtension, ' ' );
    if( pszWhere || *szTargetExtension == '\0' )
        return false;

    // 返回扩展字符串
    pszExtensions = glGetString( GL_EXTENSIONS );

    // 在扩展字符串中搜索

```

```

pszStart = pszExtensions;
for(;;)
{
    pszWhere = (unsigned char *) strstr( (const char *) pszStart, szTargetExtension );
    if( !pszWhere )
        break;
    pszTerminator = pszWhere + strlen( szTargetExtension );
    if( pszWhere == pszStart || *( pszWhere - 1 ) == ' ' )
        if( *pszTerminator == ' ' || *pszTerminator == '\0' )
            //如果存在返回True
            return true;
    pszStart = pszTerminator;
}
return false;
}

```

好了,几乎结束了,我们下面来看看我们的渲染代码.

```

void Draw (void)
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity ();

    // 显示当前的帧率
    if( GetTickCount() - g_dwLastFPS >= 1000 )
    {
        g_dwLastFPS = GetTickCount();
        g_nFPS = g_nFrames;
        g_nFrames = 0;
    }

    char szTitle[256]={0};
    sprintf( szTitle, "Lesson 45: NeHe & Paul Frazee's VBO Tut - %d Triangles, %d FPS", g_pMesh->m_nVertexCount / 3, g_nFPS );
    if( g_fVBOSupported ) // 是否支持VBO
        strcat( szTitle, ", Using VBOs" );
    else
        strcat( szTitle, ", Not Using VBOs" );
    SetWindowText( g_window->hWnd, szTitle ); // 设置窗口标题
}

```

```
}

g_nFrames++;

// 设置视口
glTranslatef( 0.0f, -220.0f, 0.0f );
glRotatef( 10.0f, 1.0f, 0.0f, 0.0f );
glRotatef( g_fLYRot, 0.0f, 1.0f, 0.0f );

// 使用顶点，纹理坐标数组
glEnableClientState( GL_VERTEX_ARRAY );
glEnableClientState( GL_TEXTURE_COORD_ARRAY );
```

为了使用VBO，你必须告诉OpenGL内存中的那部分需要加载到VBO中。所以第一步我们要起用顶点数组和纹理坐标数组。接着我们必须告诉OpenGL去把数据的指针设置到特定的地方，glVertexPointer函数可以完成这个功能。

我们分为启用和不启用VBO两个路径来渲染，他们都差不多，唯一的区别是当你需要把指针指向VBO缓存时，记得把数据指针设置NULL。

```
// 如果支持VBO扩展
if( g_fVBOSupported )
{
    glBindBufferARB( GL_ARRAY_BUFFER_ARB, g_pMesh->m_nVBOVertices );
    glVertexPointer( 3, GL_FLOAT, 0, (char *) NULL );           // 设置顶点数组的指针为顶点缓存
    glBindBufferARB( GL_ARRAY_BUFFER_ARB, g_pMesh->m_nVBOTexCoords );
    glTexCoordPointer( 2, GL_FLOAT, 0, (char *) NULL );         // 设置顶点数组的指针为纹理坐标缓存
}
// 不支持VBO扩展
else
{
    glVertexPointer( 3, GL_FLOAT, 0, g_pMesh->m_pVertices );
    glTexCoordPointer( 2, GL_FLOAT, 0, g_pMesh->m_pTexCoords );
}
```

好了,渲染所有的三角形吧

// 渲染

```
glDrawArrays( GL_TRIANGLES, 0, g_pMesh->m_nVertexCount );
```

最后,别忘了恢复到默认的OpenGL状态.

```
glDisableClientState( GL_VERTEX_ARRAY );
glDisableClientState( GL_TEXTURE_COORD_ARRAY );
```

}

如果你想更多的了解VBO对象,我建议你读一下SGI的扩展说明:

<http://oss.sgi.com/projects/ogl-sample/registry>

它会给你更多的信息

好了,那就是这次的课程,如果你发现任何问题,请联系我.

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

#### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容,大多都来自共享的



资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

感谢

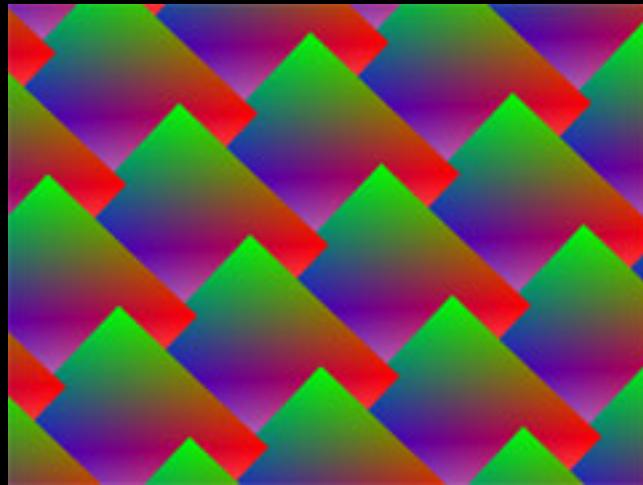
感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

< 第44课

第46课 >

## 第46课



全屏反走样

当今显卡的强大功能，你几乎什么都不用做，只需要在创建窗口的时候该一个数据。看看吧，驱动程序为你做完了一切。

在图形的绘制中，直线的走样是非常影响美观的，我们可以使用反走样解决这个问题。在众多的解决方案里，多重采样是一种易于硬件实现的方法，也是一种快速的方法。全凭多重采样可以使你的图形看起来更美观，我们可以使用ARB\_MULTISAMPLE扩展完成这个功能，但它会降低你的程序的速度。

```
Vid_mem = sizeof(Front_buffer) + sizeof(Back_buffer) + num_samples  
* (sizeof(Front_buffer) + sizeof(ZS_buffer))
```

如果你想知道更多的关于多重采样的信息 , 请访问下面的链接 :

GDC2002 -- OpenGL Multisample

OpenGL Pixel Formats and Multisample Antialiasing

下面我们来介绍如何使用多重采样 , 不向其他的扩展 , 我们在使用多重采样时 , 必须在窗口创建时告诉它使用多重采样 , 典型的步骤如下 :

- 1、 创建一个窗口
- 2、 查询是否支持多重采样
- 3、 如果支持删除当前的窗口 , 使用支持多重采样的格式创建窗口
- 4、 如果我们想使用多重采样 , 仅仅启用它既可。

了解了上面 , 我们从头说明如何使用多重采样 , 并介绍ARB\_Multisample的实现方法 :

```
#include <windows.h>
#include <gl.h>
#include <glu.h>
#include "arb_multisample.h"
```

下面两行定义我需要使用的像素格式

```
// 声明我们将要使用
#define WGL_SAMPLE_BUFFERS_ARB 0x2041
#define WGL_SAMPLES_ARB      0x2042

bool arbMultisampleSupported = false;
int arbMultisampleFormat = 0;
```

下面这个函数在扩展名的字符串中查找 , 如果包含则返回true

```
// 判断是否支持这个扩展
bool WGLIsExtensionSupported(const char *extension)
{
    const size_t extlen = strlen(extension);
    const char *supported = NULL;

// 返回在WGL的扩展中查找是否支持特定的扩展
PROC wglGetExtString = wglGetProcAddress("wglGetExtensionsStringARB");

if (wglGetExtString)
    supported = ((char*(_stdcall*)(HDC))wglGetExtString)(wglGetCurrentDC());

// 在OpenGL的扩展中查找是否支持特定的扩展
if (supported == NULL)
    supported = (char*)glGetString(GL_EXTENSIONS);

// 如果都不支持，则返回失败
if (supported == NULL)
    return false;

// 查找是否包含需要的扩展名
for (const char* p = supported; ; p++)
{
    p = strstr(p, extension);

    if (p == NULL)
        return false;

    if ((p==supported || p[-1]==' ') && (p[extlen]=='\0' || p[extlen]==' '))
        return true;
}
}
```

下面这个函数在扩展名的字符串中查找，如果包含则返回true

```
// 初始化多重渲染
bool InitMultisample(HINSTANCE hInstance,HWND hWnd,PIXELFORMATDESCRIPTOR pfd)
{
    // 检测是否支持多重渲染
    if (!WGLIsExtensionSupported("WGL_ARB_multisample"))
    {
        arbMultisampleSupported=false;
        return false;
    }

// 返回wglChoosePixelFormatARB函数的入口
PFNWGLCHOOSEPIXELFORMATARBPROC wglChoosePixelFormatARB =
(PFNWGLCHOOSEPIXELFORMATARBPROC)wglGetProcAddress("wglChoosePixelFormatARB");
if (!wglChoosePixelFormatARB)
{
    arbMultisampleSupported=false;
    return false;
}

HDC hDC = GetDC(hWnd);

int pixelFormat;
int valid;
UINT numFormats;
float fAttributes[] = {0,0};

//下面的代码设置多重采样的像素格式
int iAttributes[] =
{
    WGL_DRAW_TO_WINDOW_ARB,GL_TRUE,
    WGL_SUPPORT_OPENGL_ARB,GL_TRUE,
    WGL_ACCELERATION_ARB,WGL_FULL_ACCELERATION_ARB,
    WGL_COLOR_BITS_ARB,24,
    WGL_ALPHA_BITS_ARB,8,
    WGL_DEPTH_BITS_ARB,16,
    WGL_STENCIL_BITS_ARB,0,
    WGL_DOUBLE_BUFFER_ARB,GL_TRUE,
    WGL_SAMPLE_BUFFERS_ARB,GL_TRUE,
    WGL_SAMPLES_ARB,4,
    0,0
};
```

```
// 首先我们测试是否支持4个采样点的多重采样
valid = wglChoosePixelFormatARB(hDC,iAttributes,fAttributes,1,&pixelFormat,&numFormats);
// 如果返回true并且numformats大于1，则表示成功，那么起用多重采样
if (valid && numFormats >= 1)
{
    arbMultisampleSupported = true;
    arbMultisampleFormat = pixelFormat;
    return arbMultisampleSupported;
}

// 接着我们测试是否支持2个采样点的多重采样
iAttributes[19] = 2;
// 如果返回true并且numformats大于1，则表示成功，那么起用多重采样
valid = wglChoosePixelFormatARB(hDC,iAttributes,fAttributes,1,&pixelFormat,&numFormats);
if (valid && numFormats >= 1)
{
    arbMultisampleSupported = true;
    arbMultisampleFormat = pixelFormat;
    return arbMultisampleSupported;
}

// 返回支持多重采样
return arbMultisampleSupported;
}
```

下面到了我们的主程序部分了，和前面一样还是按照常规包含一些头文件

```
#include <windows.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include "NeHeGL.h"

#include "ARB_MULTISAMPLE.h"

BOOL DestroyWindowGL (GL_Window* window);
BOOL CreateWindowGL (GL_Window* window);
```

下面我们需要在CreateWindowGL函数中添加下面的代码，首先我们先创建一个不需要支持多重采样的窗口，它在不启用多重采样的情况下起作用。

```

//如果不启用多重采样
if(!arbMultisampleSupported)
{
    PixelFormat = ChoosePixelFormat (window->hDC, &pf);
                                // 选择一种相容的像素
格式
    if (PixelFormat == 0)
                                // 是否获得相容的像素格
式
    {
        ReleaseDC (window->hWnd, window->hDC);
                                // 释放设备描述表
        window->hDC = 0;
                                // 设置窗口设备描述
表为0
        DestroyWindow (window->hWnd);
                                // 删除窗口
        window->hWnd = 0;
                                // 设置窗口句柄为0
        return FALSE;
                                // 返回错误
    }
}

//如果起用多重采样
else
{
    PixelFormat = arbMultisampleFormat; //设置采样格式为多重采样格式
}

```

接着测试是否支持多重采样，如果支持，初始化多重采样，并重新创建窗口

```

//检测是否支持多重采样
if(!arbMultisampleSupported && CHECK_FOR_MULTISAMPLE)
{
    //如果是，初始化多重采样
    if(InitMultisample(window->init.application->hInstance,window->hWnd,pf))
    {
        //消耗当前窗口
        DestroyWindowGL (window);
        //创建一个支持多重采样的窗口
    }
}

```

```
        return CreateWindowGL(window);
    }
}
```

好了创建好多重采样后，只需要起用它。其他的一切绘制都和平常一样。

```
glEnable(GL_MULTISAMPLE_ARB);
```

// 渲染场景

```
glDisable(GL_MULTISAMPLE_ARB);
```

好了，那就是全部，希望你能喜欢：）

#### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

#### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

#### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内



容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

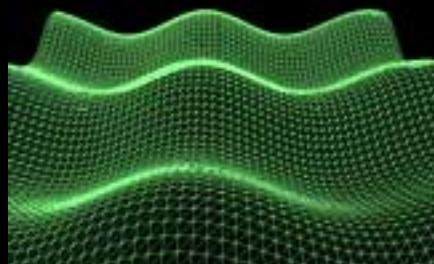
&lt; 第45课

第47课 &gt;

## 第47课



### CG 顶点脚本



nVidio的面向GPU的C语言，如果你相信它就好好学学吧，同样这里也只是个入门。记住，类似的语言还有微软的HLSL,OpenGL的GLSL,ATI的shaderMonker。不要选错哦:)

使用顶点和片断脚本去做渲染工作可以得到额外的好处，最大的好处就是把CPU的一些工作交给了GPU，Cg提供了书写这些强大的脚本的一种手段。

这篇教程有许多目的，第一向你展现了一个非常简单的顶点脚本，第二向你说明如何在OpenGL中使用Cg编写的脚本。

这个教程是基于最新的NeHeGL的基本代码，为了获得更多的信息，你可以访问nVidia的官方网站([developer.nvidia.com](http://developer.nvidia.com))，它会给你一个完整的答案。

注意：这个教程不是叫你如何去写一个完整的Cg脚本，而是教你在OpenGL中载入并运行脚本。

**开始：**

第一步，从nVidia的网站上下载Cg Compiler库，最好去下载1.1版本的，因为nvidia各个版本的变化很大，为了让程序不出现任何问题，最好这样做，因为我们用的是1.1版本的。

下一步，包含编译需要的头文件和库文件。

我已经帮你把它们拷贝到了工程的文件夹里了。

### Cg介绍

你必须有以下几个概念：

1、顶点脚本会作用于你输入的每一个顶点，如果你想要作用于一些顶点，那么你必须在作用前加载顶点脚本，并于作用后释放顶点脚本。

2、顶点脚本输出的结果被送入到片断处理器中，你不用管这其中是如何实现的。

最后，记住顶点脚本在图元装配前被执行，片断脚本在光栅化后被执行。

好了，现在我们创建一个空白的文件吧(保存为wave.cg)，接着我们创建一个数据结构，它被我们的脚本使用。下面的代码被加入到wave.cg文件中。

```
struct appdata
{
    float4 position : POSITION;
    float4 color   : COLOR0;
    float3 wave    : COLOR1;
};
```

上面的结果说明，我们输入的顶点包含一个位置坐标，一个颜色和我们自定义的波的颜色

下面的代码定义一个输出顶点的数据，包括一个顶点和颜色

```
struct vfconn
{
    float4 HPos  : POSITION;
    float4 Col0 : COLOR0;
};
```

下面的代码是Cg的主函数,每个顶点都会被以下函数执行:

```
vfconn main(appdata IN, uniform float4x4 ModelViewProj)
{
    vfconn OUT; // 保存我们输出顶点的数据

    // 计算顶点y的坐标
    IN.position.y = ( sin(IN.wave.x + (IN.position.x / 5.0) ) + sin(IN.wave.x + (IN.position.z / 4.0) ) ) * 2.5f;

    // 保存到输出数据中
    OUT.HPos = mul(ModelViewProj, IN.position);

    // 不改变输入的颜色
    OUT.Col0.xyz = IN.color.xyz;

    return OUT;
}
```

完成了上面的代码,记得保存一下.

下面我们到了程序中,首先包含使用cg需要的头文件,和库文件

```
#include <cg\cg.h>
#include <cg\cggl.h>

#pragma comment( lib, "cg.lib" )
#pragma comment( lib, "cggl.lib" )
```

下面我们定义一些全局变量 , 用来计算我们得网格和控制cg程序的开关

```
#define     SIZE  64           // 定义网格的大小
bool      cg_enable = TRUE, sp; // 开关Cg程序
GLfloat   mesh[SIZE][SIZE][3]; // 保存我们的网格
GLfloat   wave_movement = 0.0f; // 记录波动的移动
```

下面我们来定义一些cg相关的全局变量

```
CGcontext  cgContext;          // 用来保存cg脚本
```

我们需要的第一个变量是CGcontext,这个变量是多个Cg脚本的容器,一般来说,你获得你可以用函数从这个容器中获得你想要的脚本

接下来我们定义一个CGprogram变量,它用来保存我们得顶点脚本

```
CGprogram  cgProgram;         // 我们得顶点脚本
```

接下来我们需要一个变量来设置如何编译这个顶点脚本

```
CGprofile   cgVertexProfile;  // 被顶点脚本使用
```

下面我们需要一些参数用来把Cg脚本使用的数据从程序中传送过去。

```
CGparameter position, color, modelViewMatrix, wave; // 脚本中需要的参数
```

在初始化阶段我们先要创建我们网格数据

```
glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);

for (int x = 0; x < SIZE; x++)
{
for (int z = 0; z < SIZE; z++)
{
mesh[x][z][0] = (float) (SIZE / 2) - x;
mesh[x][z][1] = 0.0f;
mesh[x][z][2] = (float) (SIZE / 2) - z;
}
}
```

我们设置多边形的现实模式为线框图，接着遍历没有顶点，设置其高度。

接下来，我们初始化Cg程序

```
// 设置Cg
cgContext = cgCreateContext(); // 创建一个Cg容器

// 测试是否创建成功
if (cgContext == NULL)
{
MessageBox(NULL, "Failed To Create Cg Context", "Error", MB_OK);
return FALSE;
}
```

我们创建一个Cg程序的容器，并检查它是否创建成功

```
cgVertexProfile = cgGLGetLatestProfile(CG_GL_VERTEX); // 配置在OpenGL中使用顶点  
缓存  
  
// 检测Cg程序的是否创建成功  
if (cgVertexProfile == CG_PROFILE_UNKNOWN)  
{  
    MessageBox(NULL, "Invalid profile type", "Error", MB_OK);  
    return FALSE;  
}  
  
cgGLSetOptimalOptions(cgVertexProfile); // 启用配置文件
```

如果你想使用片断脚本，使用CG\_GL\_FRAGMENT变量。如果返回的变量为CG\_PROFILE\_UNKNOW表示没有可用的配置文件，则不能编译你需要的Cg程序。

```
// 从文件中载入Cg程序  
cgProgram = cgCreateProgramFromFile(cgContext, CG_SOURCE, "CG/Wave.cg", cgVertexProfile,  
"main", 0);
```

```
// 检测是否成功  
if (cgProgram == NULL)  
{  
    CGerror Error = cgGetError();  
    MessageBox(NULL, cgGetErrorString(Error), "Error", MB_OK);  
    return FALSE;  
}
```

我们尝试从源文件中创建一个Cg程序，并返回编译后的结果。

```
// 载入脚本  
cgGLLoadProgram(cgProgram);
```

下面我们将顶点脚本载入到显存，并准备绑定给GPU。所有的脚本在使用前必须加载。

```
// 把数据变量地址发送给Cg程序  
position = cgGetNamedParameter(cgProgram, "IN.position");  
color = cgGetNamedParameter(cgProgram, "IN.color");  
wave = cgGetNamedParameter(cgProgram, "IN.wave");  
modelViewMatrix = cgGetNamedParameter(cgProgram, "ModelViewProj");  
  
return TRUE;
```

在初始化的最后，我们必须告诉Cg脚本在那里去获得输入的数据，我们需要把变量的指针传递过去，下面的函数完成了这个功能。

程序结束时，记得释放我们创建的内容。

```
cgDestroyContext(cgContext);
```

下面的代码使用空格切换是否使用Cg程序

```
if (g_keys->keyDown [' '] && !sp)  
{  
    sp=TRUE;  
    cg_enable=!cg_enable;  
}  
  
if (!g_keys->keyDown [' '])  
    sp=FALSE;
```

现在我们已经完成了所有的准备工作了，到了我们实际绘制网格的地方了，按照惯例我们还是先设置我们得视口。

```
gluLookAt(0.0f, 25.0f, -45.0f, 0.0f, 0.0f, 0.0f, 0, 1, 0);
```

// 把当前Cg程序的模型变化矩阵告诉当前程序

```
cgGLSetStateMatrixParameter(modelViewMatrix, CG_GL_MODELVIEW_PROJECTION_MATRIX,  
CG_GL_MATRIX_IDENTITY);
```

上面我们要做的事就是把当前Cg程序的模型变化矩阵告诉当前程序。

结下来如果使用cg程序，则把顶点的颜色设置为绿色

// 如果使用Cg程序

```
if (cg_enable)  
{  
    // 使用顶点脚本配置文件  
    cgGLEnableProfile(cgVertexProfile);  
    // 帮定到当前的顶点脚本  
    cgGLBindProgram(cgProgram);  
    // 设置绘制颜色  
    cgGLSetParameter4f(color, 0.5f, 1.0f, 0.5f, 1.0f);  
}
```

下面我们来绘制我们的网格。

// 开始绘制我们的网格

```
for (int x = 0; x < SIZE - 1; x++)  
{  
    glBegin(GL_TRIANGLE_STRIP);  
}
```

```
for (int z = 0; z < SIZE - 1; z++)  
{  
    // 设置Wave参数  
    cgGLSetParameter3f(wave, wave_movement, 1.0f, 1.0f);  
    //设置输入的顶点  
    glVertex3f(mesh[x][z][0], mesh[x][z][1], mesh[x][z][2]);  
    glVertex3f(mesh[x+1][z][0], mesh[x+1][z][1], mesh[x+1][z][2]);  
    wave_movement += 0.00001f;  
    if (wave_movement > TWO_PI)  
        wave_movement = 0.0f;  
}  
//经过Cg程序处理，进行绘制  
glEnd();  
}
```

上面的代码完成具体的绘制操作，对于每一个顶点，我们动态的传入波动系数和输入原始的顶点数据。在绘制开始前，顶点脚本接受所有的顶点数据并处理，接着进行光栅化操作。

别忘了在绘制完成后，关闭我们启用的顶点脚本，否则在绘制其它的模型时会让你得到不想要的结果。

```
if (cg_enable)  
    cgGLDisableProfile(cgVertexProfile); // 禁用顶点脚本配置文件
```

好了上面就是所有的内容了,简单吧.Cg就是这么简单

### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

### 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。





## 第48课



Productions!



轨迹球实现的鼠标旋转

使用鼠标旋转物体,很简单也有很多实现方法,这里我们教会你模拟轨迹球来实现它.

### 轨迹球控制

By Terence J. Grant ([tjgrant@tatewake.com](mailto:tjgrant@tatewake.com))

如果只用鼠标来控制你的模型是不是很酷?轨迹球可以帮你做到这一点，我将告诉你我的实现，你可以把它应用在你的工程里。

我的实现是基于Bretton Wade ' s , 它是基于Ken Shoemake ' s 实现的，最初的版本，你可以从游戏编程指南这本图上找到。但我还是修正了一些错误，并优化了它。

轨迹球实现的内容就是把二维的鼠标点映射到三维的轨迹球，并基于它完成旋转变化。

为了完成这个设想，首先我们把鼠标坐标映射到[-1 , 1]之间，它很简单：

```
MousePt.X = ((MousePt.X / ((Width -1) / 2)) -1);  
MousePt.Y = -((MousePt.Y / ((Height -1) / 2))-1);
```

这只是为了数学上的简化，下面我们计算这个长度，如果它大于轨迹球的边界，我们将简单的把z轴设为0，否则我们把z轴设置为这个二维点映射到球面上对应的z值。

一旦我们有了两个点，就可以计算它的法向量了和旋转角了。

下面我们从构造函数开始，完整的讲解这个类：

```
ArcBall_t::ArcBall_t(GLfloat NewWidth, GLfloat NewHeight)
```

当点击鼠标时，记录点击的位置

```
void ArcBall_t::click(const Point2fT* NewPt)
```

当拖动鼠标时，记录当前鼠标的位置，并计算出旋转的量。

```
void ArcBall_t::drag(const Point2fT* NewPt, Quat4fT* NewRot)
```

如果窗口大小改变，设置鼠标移动的范围

```
void ArcBall_t::setBounds(GLfloat NewWidth, GLfloat NewHeight)
```

下面是完成计算所要用到的数据结果，都是一些矩阵和向量

```
Matrix4fT Transform = { 1.0f, 0.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f, 0.0f,
    0.0f, 0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 0.0f, 1.0f };
```

```
Matrix3fT LastRot = { 1.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 1.0f };
```

```
Matrix3fT ThisRot = { 1.0f, 0.0f, 0.0f,
    0.0f, 1.0f, 0.0f,
    0.0f, 0.0f, 1.0f };
```

```
ArcBallT ArcBall(640.0f, 480.0f);
Point2fT MousePt;
bool isClicked = false; // 是否点击鼠标
bool isRClicked = false; // 是否右击鼠标
bool isDragging = false; // 是否拖动
```

在上面定义的变量中，transform是我们获得的最终的变换矩阵，lastRot是上一次鼠标拖动得到的旋转矩阵，thisRot为这次鼠标拖动得到的旋转矩阵。

当我们点击鼠标时，创建一个单位旋转矩阵，当我们拖动鼠标时，这个矩阵跟踪鼠标的变化。

为了更新鼠标的移动范围，我们在函数ReshapeGL中加入下面一行：

```
void ReshapeGL (int width, int height)
{
    ...
    ArcBall.setBounds((GLfloat)width, (GLfloat)height); // 更新鼠标的移动范围
```

}

```
// 处理鼠标的按键操作
LRESULT CALLBACK WindowProc (HWND hWind, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
...
case WM_MOUSEMOVE:
MousePt.s.X = (GLfloat)LOWORD(lParam);
MousePt.s.Y = (GLfloat)HIWORD(lParam);
isClicked = (LOWORD(wParam) & MK_LBUTTON) ? true : false;
isRClicked = (LOWORD(wParam) & MK_RBUTTON) ? true : false;
break;

case WM_LBUTTONDOWN: isClicked = true; break;
case WM_RBUTTONDOWN: isRClicked = true; break;
case WM_LBUTTONUP: isClicked = false; break;
case WM_RBUTTONUP: isRClicked = false; break;
...
}
```

为了随着输入更新我们的的状态，在Update函数中需要处理更新参数

```
if (isRClicked) // 如果右键按下，这重置所有的变量
{
    Matrix3fSetIdentity(&LastRot);
    Matrix3fSetIdentity(&ThisRot);
    Matrix4fSetRotationFromMatrix3f(&Transform, &ThisRot);
}

if (!isDragging) // 如果没有拖动
{
if (isClicked) // 第一次按下
{
    isDragging = true; // 设置拖动为变量为true
    LastRot = ThisRot;
    ArcBall.click(&MousePt);
}
}
```

```
}  
else  
{  
if (isClicked) //如果按住拖动  
{  
Quat4fT ThisQuat;  
  
ArcBall.drag(&MousePt, &ThisQuat); // 更新轨迹球的变量  
Matrix3fSetRotationFromQuat4f(&ThisRot, &ThisQuat); // 计算旋转量  
Matrix3fMulMatrix3f(&ThisRot, &LastRot);  
Matrix4fSetRotationFromMatrix3f(&Transform, &ThisRot);  
}  
else // 如果放开鼠标，设置拖动为false  
isDragging = false;  
}
```

好了，完成了上面的内容。我们到了绘制的阶段。  
记住在绘制前，把我们得到的矩阵乘以当前的模型变换矩阵。

```
glPushMatrix(); // 保存当前的矩阵  
glMultMatrixf(Transform.M); // 应用我们的变换矩阵  
  
glBegin(GL_TRIANGLES); // 绘制模型  
...  
glEnd();  
  
glPopMatrix(); // 弹出保存的矩阵
```

我已经在上面给掩饰了所有的技巧，你可以不使用我告诉你的数学技巧，因为我想你会有更好的。现在你已经看到了，这是多么的简单，你完全可以按你的风格创造出更好的轨迹球。



### 版权与使用声明:

我是个对学习和生活充满激情的普通男孩,在网络上我以DancingWind为昵称,我的联系方式是zhouwei02@mails.tsinghua.edu.cn,如果你有任何问题,都可以联系我。

### 引子

网络是一个共享的资源,但我在自己的学习生涯中浪费大量的时间去搜索可用的资料,在现实生活中花费了大量的金钱和时间在书店中寻找资料,于是我给自己起了个昵称DancingWind,其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后,我决定把自己收集的资料整理为一个统一的资源库。

### 版权声明

所有DancingWind发表的内容,大多都来自共享的资源,所以我没有资格把它们据为己有,或声称自己为这些资源作出了一点贡献。故任何人都可以复制,修改,重新发表,甚至以自己的名义发表,我都不会追究,但你在做以上事情的时候必须保证内容的完整性,给后来的人一个完整的教程。最后,任何人不能以这些资料的任何部分,谋取任何形式的报酬。

### 发展计划

在国外,很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识,我很欢迎你与我联系,但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹,一直以来默默的在精神上和生活中对我的支持,她甚至把买衣服的钱都用来给我买书了,她真的是我见过的最好的女孩,希望我能带给她幸福。

< 第47课



## 第01课



创建一个OpenGL窗口：

在这个教程里,我将教你在Windows环境中创建OpenGL程序.它将显示一个空的OpenGL窗口,可以在窗口和全屏模式下切换,按ESC退出.它是以后应用程序的框架.

理解OpenGL如何工作非常重要，你可以在教程的末尾下载源程序，但我强烈建议你至少读一遍教程，然后再开始编程.

欢迎来到我的 OpenGL教程。我是个对 OpenGL充满激情的普通男孩！我第一次听说 OpenGL是 3Dfx 发布 Voodoo1 卡的 OpenGL硬件加速驱动的时候。我立刻意识到 OpenGL是那种必须学习的东西。不幸的是当时很难从书本或网络上找到关于 OpenGL的讯息。我花了 N 个小时来调试自己书写的代码，甚至在 IRC 和 EMail 上花更多的时间来恳求别人帮忙。但我发现那些懂得 OpenGL 高手们保留了他们的精华，对共享知识也不感兴趣。实在让人灰心！

我创建这个网站的目的是为了帮助那些对 OpenGL有兴趣却又需要帮助的人。在我的每个教程中，我都会尽可能详细的来解释每一行代码的作用。我会努力让我的代码更简单（您无需学习 MFC 代码）！就算您是个VC、OPENGL的绝对新手也应该可以读通代码，并清楚的知道发生了什么。我的站点只是许多提供 OpenGL教程的站点中的一个。如果您是 OpenGL的高级程序员的话，我的站点可能太简单了，但如果您的才开始的话，我想这个站点会教会您许多东西！

教程的这一节在2000年一月彻底重写了一遍。将会教您如何设置一个 OpenGL窗口。它可以只是一个窗口或是全屏幕的、可以任意大小、任意色彩深度。此处的代码很稳定且很强大，您可以在您所有的OpenGL项目中使用。我所有的教程都将基于此节的代码！所有的错误都有被报告。所以应该没有内存泄漏，代码也很容易阅读和修改。感谢 Fredric Echols 对代码所做的修改！

现在就让我们直接从代码开始吧。第一件事是打开VC然后创建一个新工程。如果您不知道如何创建的话，您也许不该学习OpenGL，而应该先学学VC。某些版本的VC需要将bool改成BOOL, true改成TRUE, false改成FALSE，请自行修改。

在您创建一个新的Win32程序（不是console控制台程序）后，您还需要链接OpenGL库文件。在VC中操作如下：Project->Settings,然后单击LINK标签。在"Object/Library Modules"选项中的开始处（在kernel32.lib前）增加OpenGL32.lib GLu32.lib 和 GLaux.lib 后单击OK按钮。现在可以开始写您的OpenGL程序了。

代码的前4行包括了我们使用的每个库文件的头文件。如下所示：

```
#include <windows.h>      // Windows的头文件
#include <glew.h>          // 包含最新的gl.h,glu.h库
#include <glut.h>          // 包含OpenGL实用库
```

接下来您需要设置您计划在您的程序中使用的所有变量。本节中的例程将创建一个空的OpenGL窗口，因此我们暂时还无需设置大堆的变量。余下需要设置的变量不多，但十分重要。您将会在您以后所写的每一个OpenGL程序中用到它们。

第一行设置的变量是Rendering Context(着色描述表)。每一个OpenGL都被连接到一个着色描述表上。着色描述表将所有的OpenGL调用命令连接到Device Context(设备描述表)上。我将OpenGL的着色描述表定义为hRC。要让您的程序能够绘制窗口的话，还需要创建一个设备描述表，也就是第二行的内容。Windows的设备描述表被定义为hDC。DC将窗口连接到GDI(Graphics Device Interface图形设备接口)。而RC将OpenGL连接到DC。第三行的变量hWnd将保存由Windows给我们的窗口指派的句柄。最后，第四行为我们的程序创建了一个Instance(实例)。

```
HGLRC    hRC=NULL;           // 窗口着色描述表句柄
HDC      hDC=NULL;          // OpenGL渲染描述表句柄
HWND     hWnd=NULL;          // 保存我们的窗口句柄
HINSTANCE hInstance;         // 保存程序的实例
```

下面的第一行设置一个用来监控键盘动作的数组。有许多方法可以监控键盘的动作，但这里的方法很可靠，并且可以处理多个键同时按下的情况。

active 变量用来告知程序窗口是否处于最小化的状态。如果窗口已经最小化的话，我们可以做从暂停代码执行到退出程序的任何事情。我喜欢暂停程序。这样可以使得程序不用在后台保持运行。

fullscreen 变量的作用相当明显。如果我们的程序在全屏状态下运行，fullscreen 的值为 TRUE，否则为FALSE。这个全局变量的设置十分重要，它让每个过程都知道程序是否运行在全屏状态下。

```
bool keys[256];                                // 保存键盘按键的数组
bool active=TRUE;                             // 窗口的活动标志，缺省为TRUE
bool fullscreen=TRUE;                          // 全屏标志缺省，缺省设定成全屏模式
```

现在我们需要先定义WndProc()。必须这么做的原因是CreateGLWindow()有对WndProc()的引用，但WndProc()在CreateGLWindow()之后才出现。在C语言中，如果我们想要访问一个当前程序段之后的过程和程序段的话，必须在程序开始处先申明所要访问的程序段。所以下面的一行代码先行定义了WndProc()，使得CreateGLWindow()能够引用 WndProc()。

```
LRESULT CALLBACK WndProc(HWND, UINT, WPARAM, LPARAM);           // WndProc的定义
```

下面的代码的作用是重新设置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。

```
GLvoid ReSizeGLScene(GLsizei width, GLsizei height)           // 重置OpenGL窗口大小
{
    if (height==0)                                         // 防止被零除
    {
        height=1;                                         // 将Height设为1
    }
}
```

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

// 重置当前的视口

下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。0.1f, 100.0f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。

```
glMatrixMode(GL_PROJECTION); // 选择投影矩阵
glLoadIdentity(); // 重置投影矩阵
```

// 设置视口的大小

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

```
glMatrixMode(GL_MODELVIEW); // 选择模型观察矩阵
glLoadIdentity(); // 重置模型观察矩阵
```

}

接下的代码段中，我们将对OpenGL进行所有的设置。我们将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。此过程将有返回值。但我们此处的初始化没那么复杂，现在还用不着担心这个返回值。

```
int InitGL(GLvoid) // 此处开始对OpenGL进行所有设置
{
```

下一行启用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。

```
glShadeModel(GL_SMOOTH); // 启用阴影平滑
```

下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0.0f到1.0f。0.0f代表最黑的情况，1.0f就是最亮的情况。

glClearColor 后的第一个参数是Red Intensity(红色分量),第二个是绿色，第三个是蓝色。最大值也是1.0f，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0.0f。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(0.0f,0.0f,1.0f,0.0f)，您将用亮蓝色来清除屏幕。如果您用glClearColor(0.5f,0.0f,0.0f,0.0f)的话，您将使用中红色来清除屏幕。不是最亮(1.0f)，也不是最暗(0.0f)。要得到白色背景，您应该将所有的颜色设成最亮(1.0f)。要黑色背景的话，您该将所有的颜色设为最暗(0.0f)。

```
glClearColor(0.0f, 0.0f, 0.0f, 0.0f); // 黑色背景
```

接下来的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。

```
glClearDepth(1.0f); // 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
```

接着告诉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。但使得透视线看起来好一点。

```
glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST); // 告诉系统对透视进行修正
```

最后，我们返回TRUE。如果我们希望检查初始化是否OK，我们可以查看返回的TRUE或FALSE的值。如果有错误发生的话，您可以加上您自己的代码返回FALSE。目前，我们不管它。

```
return TRUE; // 初始化 OK
}
```

下一段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，返回TRUE值之前，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。

返回TRUE值告知我们的程序没有出现问题。如果您希望程序因为某些原因而中止运行，在返回TRUE值之前增加返回FALSE的代码告知我们的程序绘图代码出错。程序即刻退出。

```
int DrawGLScene(GLvoid) // 从这里开始进行所有的绘制
{
    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
    return TRUE; // 一切 OK
}
```

下一段代码只在程序退出之前调用。KillGLWindow() 的作用是依次释放着色描述表，设备描述表和窗口句柄。我已经加入了许多错误检查。如果程序无法销毁窗口的任意部分，都会弹出带相应错误消息的讯息窗口，告诉您什么出错了。使您在您的代码中查错变得更容易些。

```
GLvoid KillGLWindow(GLvoid) // 正常销毁窗口  
{
```

我们在KillGLWindow()中所作的第一件事是检查我们是否处于全屏模式。如果是，我们要切换回桌面。我们本应在禁用全屏模式前先销毁窗口，但在某些显卡上这么做可能会使得桌面崩溃。所以我们还是先禁用全屏模式。这将防止桌面出现崩溃，并在Nvidia和3dfx显卡上都工作的很好！

```
if (fullscreen) // 我们处于全屏模式吗?  
{
```

我们使用ChangeDisplaySettings(NULL,0)回到原始桌面。将NULL作为第一个参数，0作为第二个参数传递强制Windows使用当前存放在注册表中的值(缺省的分辨率、色彩深度、刷新频率，等等)来有效的恢复我们的原始桌面。切换回桌面后，我们还要使得鼠标指针重新可见。

```
ChangeDisplaySettings(NULL,0); // 是的话，切换回桌面  
ShowCursor(TRUE); // 显示鼠标指针  
}
```

接下来的代码查看我们是否拥有着色描述表(hRC)。如果没有，程序将跳转至后面的代码查看是否拥有设备描述表。

```
if (hRC) // 我们拥有OpenGL渲染描述表吗?  
{
```

如果存在着色描述表的话，下面的代码将查看我们能否释放它(将 hRC从hDC分开)。这里请注意我使用的的查错方法。基本上我只是让程序尝试释放着色描述表(通过调用 wglGetCurrent(NULL,NULL) )，然后我再查看释放是否成功。巧妙的将数行代码结合到了一行。

```
if (!wglGetCurrent(NULL,NULL)) // 我们能否释放DC和RC描述表?
{

```

如果不能释放DC和RC描述表的话，MessageBox()将弹出错误消息，告知我们DC和RC无法被释放。NULL意味着消息窗口没有父窗口。其右的文字将在消息窗口上出现。"SHUTDOWN ERROR"出现在窗口的标题栏上。MB\_OK的意思消息窗口上带有一个写着OK字样的按钮。

MB\_ICONINFORMATION将在消息窗口中显示一个带圈的小写的i(看上去更正式一些)。

```
    MessageBox(NULL,"释放DC或RC失败。","关闭错误",MB_OK | MB_ICONINFORMATION);
}
```

下一步我们试着删除着色描述表。如果不成功的话弹出错误消息。

```
if (!wglDeleteContext(hRC)) // 我们能否删除RC?
{

```

如果无法删除着色描述表的话，将弹出错误消息告知我们RC未能成功删除。然后hRC被设为NULL。

```
    MessageBox(NULL,"释放RC失败。","关闭错误",MB_OK | MB_ICONINFORMATION);
}
hRC=NULL; // 将RC设为 NULL
```

}

现在我们查看是否存在设备描述表，如果有尝试释放它。如果不能释放设备描述表将弹出错误消息，然后hDC设为NULL。

```
if (hDC && !ReleaseDC(hWnd,hDC)) // 我们能否释放 DC?  
{  
    MessageBox(NULL,"释放DC失败。","关闭错误",MB_OK | MB_ICONINFORMATION);  
    hDC=NULL; // 将 DC 设为 NULL  
}
```

现在我们来查看是否存在窗口句柄，我们调用 DestroyWindow( hWnd )来尝试销毁窗口。如果不能的话弹出错误窗口，然后hWnd被设为NULL。

```
if (hWnd && !DestroyWindow(hWnd)) // 能否销毁窗口?  
{  
    MessageBox(NULL,"释放窗口句柄失败。","关闭错误",MB_OK |  
    MB_ICONINFORMATION);  
    hWnd=NULL; // 将 hWnd 设为 NULL  
}
```

最后要做的事是注销我们的窗口类。这允许我们正常销毁窗口，接着在打开其他窗口时，不会收到诸如"Windows Class already registered"(窗口类已注册)的错误消息。

```
if (!UnregisterClass("OpenG",hInstance)) // 能否注销类?  
{  
    MessageBox(NULL,"不能注销窗口类。","关闭错误",MB_OK | MB_ICONINFORMATION);  
    hInstance=NULL; // 将 hInstance 设为 NULL  
}
```

接下来的代码段创建我们的OpenGL窗口。我花了很多时间来做决定是否创建固定的全屏模式这样不需要许多额外的代码，还是创建一个容易定制的友好的窗口但需要更多的代码。当然最后我选择了后者。我经常在EMail中收到诸如此类的问题：怎样创建窗口而不使用全屏幕？怎样改变窗口的标题栏？怎样改变窗口的分辨率或pixel format(象素格式)？以下的代码完成了所有这一切！尽管最好要学学材质，这会让您写自己的OpenGL程序变得容易的多！

正如您所见，此过程返回布尔变量(TRUE 或 FALSE)。他还带有5个参数：窗口的标题栏，窗口的宽度，窗口的高度，色彩位数(16/24/32)，和全屏标志(TRUE --全屏模式，FALSE--窗口模式)。返回的布尔值告诉我们窗口是否成功创建。

```
BOOL CreateGLWindow(char* title, int width, int height, int bits, bool fullscreenflag)
{
```

当我们要求Windows为我们寻找相匹配的象素格式时，Windows寻找结束后将模式值保存在变量PixelFormat中。

```
GLuint      PixelFormat;           // 保存查找匹配的结果
```

wc用来保存我们的窗口类的结构。窗口类结构中保存着我们的窗口信息。通过改变类的不同字段我们可以改变窗口的外观和行为。每个窗口都属于一个窗口类。当您创建窗口时，您必须为窗口注册类。

```
WNDCLASS    wc;           // 窗口类结构
```

dwExStyle和dwStyle存放扩展和通常的窗口风格信息。我使用变量来存放风格的目的是为了能够根据我需要创建的窗口类型(是全屏幕下的弹出窗口还是窗口模式下的带边框的普通窗口)；来改变窗口的风格。

```

DWORD      dwExStyle;           // 扩展窗口风格
DWORD      dwStyle;            // 窗口风格

```

下面的5行代码取得矩形的左上角和右下角的坐标值。我们将使用这些值来调整我们的窗口使得其上的绘图区的大小恰好是我们所需的分辨率的值。通常如果我们创建一个640x480的窗口，窗口的边框会占掉一些分辨率的值。

```

RECT WindowRect;
WindowRect.left=(long)0;          // 取得矩形的左上角和右下角的坐标值
WindowRect.right=(long)width;     // 将Left 设为0
WindowRect.top=(long)0;           // 将Right 设为要求的宽度
WindowRect.bottom=(long)height;    // 将Top 设为0
                                  // 将Bottom 设为要求的高度

```

下一行代码我们让全局变量fullscreen等于fullscreenflag。如果我们希望在全屏幕下运行而将fullscreenflag设为TRUE，但没有让变量fullscreen等于fullscreenflag的话，fullscreen变量将保持为FALSE。当我们在全屏幕模式下销毁窗口的时候，变量fullscreen的值却不是正确的TRUE值，计算机将误以为已经处于桌面模式而无法切换回桌面。上帝啊，但愿这一切都有意义。就是一句话，fullscreen的值必须永远fullscreenflag的值，否则就会有问题。

```
fullscreen=fullscreenflag;          // 设置全局全屏标志
```

下一部分的代码中，我们取得窗口的实例，然后定义窗口类。

CS\_HREDRAW 和 CS\_VREDRAW 的意思是无论何时，只要窗口发生变化时就强制重画。CS\_OWNDC为窗口创建一个私有的DC。这意味着DC不能在程序间共享。WndProc是我们程序的消息处理过程。由于没有使用额外的窗口数据，后两个字段设为零。然后设置实例。接着我们将hIcon设为NULL，因为我们不想给窗口来个图标。鼠标指针设为标准的箭头。背景色无所谓(我们在GL中设置)。我们也不想要窗口菜单，所以将其设为NULL。类的名字可以您想要的任何名字。出于简单，我将使用"OpenG"。

```

hInstance      = GetModuleHandle(NULL);           // 取得我们窗口的实例
wc.style       = CS_HREDRAW | CS_VREDRAW | CS_OWNDC; // 移动时重画，并为窗

```

## 口取得DC

```

wc.lpfnWndProc    = (WNDPROC) WndProc;           // WndProc处理消息
wc.cbClsExtra     = 0;                            // 无额外窗口数据
wc.cbWndExtra     = 0;                            // 无额外窗口数据
wc.hInstance       = hInstance;                   // 设置实例
wc.hIcon          = LoadIcon(NULL, IDI_WINLOGO); // 装入缺省图标
wc.hCursor         = LoadCursor(NULL, IDC_ARROW); // 装入鼠标指针
wc.hbrBackground   = NULL;                         // GL不需要背景
wc.lpszMenuName   = NULL;                         // 不需要菜单
wc.lpszClassName  = "OpenG";                      // 设定类名字

```

现在注册类名字。如果有错误发生，弹出错误消息窗口。按下上面的OK按钮后，程序退出

```

if (!RegisterClass(&wc))                         // 尝试注册窗口类
{
    MessageBox(NULL,"注册窗口失败","错误",MB_OK|MB_ICONEXCLAMATION);
    return FALSE;                                // 退出并返回FALSE
}

```

查看程序应该在全屏模式还是窗口模式下运行。如果应该是全屏模式的话，我们将尝试设置全屏模式。

```

if (fullscreen)                                  // 要尝试全屏模式吗?
{

```

下一部分的代码看来很多人都会有问题要问关于.....切换到全屏模式。在切换到全屏模式时，有几件十分重要的事您必须牢记。必须确保您在全屏模式下所用的宽度和高度等同于窗口模式下的宽度和高度。最最重要的是要在创建窗口之前设置全屏模式。这里的代码中，您无需再担心宽度和高度，它们已被设置成与显示模式所对应的大小。

```

DEVMODE dmScreenSettings;                      // 设备模式

```

```

memset(&dmScreenSettings,0,sizeof(dmScreenSettings));           // 确保内存清空为零
dmScreenSettings.dmSize = sizeof(dmScreenSettings);           // Devmode 结构的大小
dmScreenSettings.dmPelsWidth = width;                         // 所选屏幕宽度
dmScreenSettings.dmPelsHeight = height;                        // 所选屏幕高度
dmScreenSettings.dmBitsPerPel = bits;                          // 每象素所选的色彩深度
dmScreenSettings.dmFields=DM_BITSPERPEL|DM_PELSWIDTH|DM_PELSHEIGHT;

```

上面的代码中，我们分配了用于存储视频设置的空间。设定了屏幕的宽，高，色彩深度。下面的代码我们尝试设置全屏模式。我们在dmScreenSettings中保存了所有的宽，高，色彩深度讯息。下一行使用ChangeDisplaySettings来尝试切换成与dmScreenSettings所匹配模式。我使用参数CDS\_FULLSCREEN来切换显示模式，因为这样做不仅移去了屏幕底部的状态条，而且它在来回切换时，没有移动或改变您在桌面上的窗口。

```

// 尝试设置显示模式并返回结果。注: CDS_FULLSCREEN 移去了状态条。
if (ChangeDisplaySettings(&dmScreenSettings,CDS_FULLSCREEN) != DISP_CHANGE_SUCCESSFUL)
{

```

如果模式未能设置成功，我们将进入以下的代码。如果不能匹配全屏模式，弹出消息窗口，提供两个选项：在窗口模式下运行或退出。

```

// 若模式失败，提供两个选项：退出或在窗口内运行。
if (MessageBox(NULL,"全屏模式在当前显卡上设置失败！\n使用窗口模式？","NeHe G",
MB_YESNO|MB_ICONEXCLAMATION)==IDYES)
{

```

如果用户选择窗口模式，变量fullscreen 的值变为FALSE,程序继续运行。

```

fullscreen=FALSE;                                // 选择窗口模式(Fullscreen=FALSE)
}
else
{

```

如果用户选择退出，弹出消息窗口告知用户程序将结束。并返回FALSE告诉程序窗口未能成功创建。程序退出。

```
// 弹出一个对话框，告诉用户程序结束  
MessageBox(NULL,"程序将被关闭","错误",MB_OK|MB_ICONSTOP);  
return FALSE; // 退出并返回 FALSE  
}  
}  
}
```

由于全屏模式可能失败，用户可能决定在窗口下运行，我们需要在设置屏幕/窗口之前，再次检查fullscreen的值是TRUE或FALSE。

```
if (fullscreen) // 仍处于全屏模式吗?  
{
```

如果我们仍处于全屏模式，设置扩展窗体风格为WS\_EX\_APPWINDOW，这将强制我们的窗体可见时处于最前面。再将窗体的风格设为WS\_POPUP。这个类型的窗体没有边框，使我们的全屏模式得以完美显示。

最后我们禁用鼠标指针。当您的程序不是交互式的时候，在全屏模式下禁用鼠标指针通常是个好主意。

```
dwExStyle=WS_EX_APPWINDOW; // 扩展窗体风格  
dwStyle=WS_POPUP; // 窗体风格  
ShowCursor(FALSE); // 隐藏鼠标指针  
}  
else  
{
```

如果我们使用窗口而不是全屏模式，我们在扩展窗体风格中增加了 WS\_EX\_WINDOWEDGE，增强窗体的3D感观。窗体风格改用 WS\_OVERLAPPEDWINDOW，创建一个带标题栏、可变大小的边框、菜单和最大化/最小化按钮的窗体。

```
dwExStyle=WS_EX_APPWINDOW | WS_EX_WINDOWEDGE;           // 扩展窗体风格
dwStyle=WS_OVERLAPPEDWINDOW;                            // 窗体风格
}
```

下一行代码根据创建的窗体类型调整窗口。调整的目的是使得窗口大小正好等于我们要求的分辨率。通常边框会占用窗口的一部分。使用AdjustWindowRectEx后，我们的OpenGL场景就不会被边框盖住。实际上窗口变得更大以便绘制边框。全屏模式下，此命令无效。

```
AdjustWindowRectEx(&WindowRect, dwStyle, FALSE, dwExStyle); // 调整窗口达到真正要求的大小
```

下一段代码开始创建窗口并检查窗口是否成功创建。我们将传递CreateWindowEx()所需的所有参数。如扩展风格、类名字(与您在注册窗口类时所用的名字相同)、窗口标题、窗体风格、窗体的左上角坐标(0,0 是个安全的选择)、窗体的宽和高。我们没有父窗口，也不想要菜单，这些参数被设为NULL。还传递了窗口的实例，最后一个参数被设为NULL。

注意我们在窗体风格中包括了 WS\_CLIPSIBLINGS 和 WS\_CLIPCHILDREN。要让OpenGL正常运行，这两个属性是必须的。他们阻止别的窗体在我们的窗体内/上绘图。

```
if (!(hWnd=CreateWindowEx(  dwExStyle,           // 扩展窗体风格
                           "OpenG",            // 类名字
                           title,              // 窗口标题
                           WS_CLIPSIBLINGS |   // 必须的窗体风格属性
                           WS_CLIPCHILDREN |  // 必须的窗体风格属性
                           dwStyle,             // 选择的窗体属性
                           0, 0,                // 窗口位置
                           WindowRect.right-WindowRect.left, // 计算调整好的窗口宽度
```

```

WindowRect.bottom-WindowRect.top, // 计算调整好的窗口高度
NULL, // 无父窗口
NULL, // 无菜单
hInstance, // 实例
NULL)) // 不向WM_CREATE传递任何东东

```

下来我们检查看窗口是否正常创建。如果成功，hWnd保存窗口的句柄。如果失败，弹出消息窗口，并退出程序。

```

{
    KillGLWindow(); // 重置显示区
    MessageBox(NULL,"不能创建一个窗口设备描述表","错误",MB_OK|MB_ICONEXCLAMATION);
    return FALSE; // 返回 FALSE
}

```

下面的代码描述像素格式。我们选择了通过RGBA(红、绿、蓝、alpha通道)支持OpenGL和双缓存的格式。我们试图找到匹配我们选定的色彩深度(16位、24位、32位)的像素格式。最后设置16位Z-缓存。其余的参数要么未使用要么不重要(stencil buffer模板缓存和accumulation buffer聚集缓存除外)。

```

static PIXELFORMATDESCRIPTOR pfd= // /pfд告诉窗口我们所希望的东西，即窗口使用的像素格式
{
    sizeof(PIXELFORMATDESCRIPTOR), // 上述格式描述符的大小
    1, // 版本号
    PFD_DRAW_TO_WINDOW | // 格式支持窗口
    PFD_SUPPORT_OPENGL | // 格式必须支持OpenGL
    PFD_DOUBLEBUFFER, // 必须支持双缓冲
    PFD_TYPE_RGBA, // 申请 RGBA 格式
    bits, // 选定色彩深度
    0, 0, 0, 0, 0, 0, // 忽略的色彩位
    0, // 无Alpha缓存
    0, // 忽略Shift Bit
    0, // 无累加缓存
    0, 0, 0, 0, 0, 0 // 忽略聚集位
}

```

```

16,                                // 16位 Z-缓存(深度缓存)
0,                                 // 无蒙板缓存
0,                                 // 无辅助缓存
PFD_MAIN_PLANE,                  // 主绘图层
0,                                 // Reserved
0, 0, 0                           // 忽略层遮罩
};


```

如果前面创建窗口时没有错误发生，我们接着尝试取得OpenGL设备描述表。若无法取得DC，弹出错误消息程序退出(返回FALSE)。

```

if (!(hDC=GetDC(hWnd)))           // 取得设备描述表了么?
{
    KillGLWindow();               // 重置显示区
    MessageBox(NULL,"不能创建一种相匹配的像素格式","错误",MB_OK|MB_ICONEXCLAMATION);
    return FALSE;                 // 返回 FALSE
}


```

设法为OpenGL窗口取得设备描述表后，我们尝试找到对应与此前我们选定的像素格式的像素格式。如果Windows不能找到的话，弹出错误消息，并退出程序(返回FALSE)。

```

if (!(PixelFormat=ChoosePixelFormat(hDC,&pf)))           // Windows 找到相应的像素格式了吗?
{
    KillGLWindow();               // 重置显示区
    MessageBox(NULL,"不能设置像素格式","错误",MB_OK|MB_ICONEXCLAMATION);
    return FALSE;                 // 返回 FALSE
}


```

Windows 找到相应的像素格式后，尝试设置像素格式。如果无法设置，弹出错误消息，并退出程序(返回FALSE)。

```

if(!SetPixelFormat(hDC,PixelFormat,&pf))
    // 能够设置像素格式么?
{
    KillGLWindow();           // 重置显示区
    MessageBox(NULL,"不能设置像素格式","错误",MB_OK|MB_ICONEXCLAMATION);
    return FALSE;             // 返回 FALSE
}

```

正常设置像素格式后，尝试取得着色描述表。如果不能取得着色描述表的话，弹出错误消息，并退出程序(返回FALSE)。

```

if (!(hRC=wglCreateContext(hDC)))          // 能否取得着色描述表?
{
    KillGLWindow();           // 重置显示区
    MessageBox(NULL,"不能创建OpenGL渲染描述表","错误",MB_OK|
MB_ICONEXCLAMATION);
    return FALSE;             // 返回 FALSE
}

```

如果到现在仍未出现错误的话，我们已经设法取得了设备描述表和着色描述表。接着要做的是激活着色描述表。如果无法激活，弹出错误消息，并退出程序(返回FALSE)。

```

if(!wglGetCurrent(hDC,hRC))                // 尝试激活着色描述表
{
    KillGLWindow();           // 重置显示区
    MessageBox(NULL,"不能激活当前的OpenGL渲染描述表","错误",MB_OK|
MB_ICONEXCLAMATION);
    return FALSE;             // 返回 FALSE
}

```

一切顺利的话，OpenGL窗口已经创建完成，接着可以显示它啦。将它设为前端窗口(给它更高的优先级)，并将焦点移至此窗口。然后调用ResizeGLScene 将屏幕的宽度和高度设置给透视OpenGL屏幕。

```

ShowWindow(hWnd,SW_SHOW);           // 显示窗口
SetForegroundWindow(hWnd);          // 略略提高优先级
SetFocus(hWnd);                   // 设置键盘的焦点至此窗口
ReSizeGLScene(width, height);      // 设置透视 GL 屏幕

```

跳转至 InitGL()，这里可以设置光照、纹理、等等任何需要设置的东东。您可以在 InitGL() 内部自行定义错误检查，并返回 TRUE (一切正常) 或 FALSE (有什么不对)。例如，如果您在 InitGL() 内装载纹理并出现错误，您可能希望程序停止。如果您返回 FALSE 的话，下面的代码会弹出错误消息，并退出程序。

```

if (!InitGL())
{
    KillGLWindow();           // 重置显示区
    MessageBox(NULL,"Initialization Failed.","ERROR",MB_OK|MB_ICONEXCLAMATION);
    return FALSE;             // 返回 FALSE
}

```

到这里可以安全的推定创建窗口已经成功了。我们向 WinMain() 返回 TRUE，告知 WinMain() 没有错误，以防止程序退出。

```

return TRUE;                  // 成功
}

```

下面的代码处理所有的窗口消息。当我们注册好窗口类之后，程序跳转到这部分代码处理窗口消息。

```

LRESULT CALLBACK WndProc(    HWND hWnd,           // 窗口的句柄
                           UINT uMsg,          // 窗口的消息
                           WPARAM wParam,       // 附加的消息内容
                           LPARAM lParam)       // 附加的消息内容
{

```

下面的代码比对uMsg的值，然后转入case处理，uMsg中保存了我们要处理的消息名字。

```
switch (uMsg) // 检查Windows消息
{
```

如果uMsg等于WM\_ACTIVE，查看窗口是否仍然处于激活状态。如果窗口已被最小化，将变量active设为FALSE。如果窗口已被激活，变量active的值为TRUE。

```
case WM_ACTIVATE: // 监视窗口激活消息
{
    if (!HIWORD(wParam)) // 检查最小化状态
    {
        active=TRUE; // 程序处于激活状态
    }
    else
    {
        active=FALSE; // 程序不再激活
    }
}
return 0; // 返回消息循环
}
```

如果消息是WM\_SYSCOMMAND(系统命令)，再次比对wParam。如果wParam是SC\_SCREENSAVE或SC\_MONITORPOWER的话，不是有屏幕保护要运行，就是显示器想进入节电模式。返回0可以阻止这两件事发生。

```
case WM_SYSCOMMAND: // 系统中断命令
{
    switch (wParam) // 检查系统调用
    {
        case SC_SCREENSAVE: // 屏保要运行?
        case SC_MONITORPOWER: // 显示器要进入节电模式?
```

```

        return 0;           // 阻止发生
    }
    break;                // 退出
}

```

如果 uMsg是WM\_CLOSE，窗口将被关闭。我们发出退出消息，主循环将被中断。变量 done被设为TRUE，WinMain()的主循环中止，程序关闭。

```

case WM_CLOSE:           // 收到Close消息?
{
    PostQuitMessage(0);   // 发出退出消息
    return 0;              // 返回
}

```

如果键盘有键按下，通过读取wParam的信息可以找出键值。我将键盘数组keys[ ]相应的数组组成员的值设为TRUE。这样以后就可以查找key[ ]来得知什么键被按下。允许同时按下多个键。

```

case WM_KEYDOWN:         // 有键按下么?
{
    keys[wParam] = TRUE;   // 如果是，设为TRUE
    return 0;                  // 返回
}

```

同样，如果键盘有键释放，通过读取wParam的信息可以找出键值。然后将键盘数组keys[ ]相应的数组组成员的值设为FALSE。这样查找key[ ]来得知什么键被按下，什么键被释放了。键盘上的每个键都可以用0-255之间的一个数来代表。举例来说，当我们按下40所代表的键时，keys[40]的值将被设为TRUE。放开的话，它就被设为FALSE。这也是key数组的原理。

```

case WM_KEYUP:           // 有键放开么?
{

```

```

    keys[wParam] = FALSE;           // 如果是，设为FALSE
    return 0;                      // 返回
}

```

当调整窗口时，uMsg 最后等于消息WM\_SIZE。读取IParam的LOWORD 和HIWORD可以得到窗口新的宽度和高度。将他们传递给ReSizeGLScene()，OpenGL场景将调整为新的宽度和高度。

```

case WM_SIZE:                  // 调整OpenGL窗口大小
{
    ReSizeGLScene(LOWORD(IParam),HIWORD(IParam));    // LoWord=Width,
    HiWord=Height
    return 0;                                         // 返回
}
}

```

其余无关的消息被传递给DefWindowProc，让Windows自行处理。

```

// 向 DefWindowProc传递所有未处理的消息。
return DefWindowProc(hWnd,uMsg,wParam,IParam);
}

```

下面是我们的Windows程序的入口。将会调用窗口创建例程，处理窗口消息，并监视人机交互。

```

int WINAPI WinMain( HINSTANCE hInstance,           // 当前窗口实例
                     HINSTANCE hPrevInstance,      // 前一个窗口实例
                     LPSTR     lpCmdLine,         // 命令行参数
                     int       nCmdShow)          // 窗口显示状态
{

```

我们设置两个变量。msg 用来检查是否有消息等待处理。done的初始值设为FALSE。这意味着我们的程序仍未完成运行。只要程序done保持FALSE，程序继续运行。一旦done的值改变为TRUE，程序退出。

```
MSG msg; // Windows消息结构
BOOL done=FALSE; // 用来退出循环的Bool 变量
```

这段代码完全可选。程序弹出一个消息窗口，询问用户是否希望在全屏模式下运行。如果用户单击NO按钮，fullscreen变量从缺省的TRUE改变为FALSE，程序也改在窗口模式下运行。

```
// 提示用户选择运行模式
if (MessageBox(NULL,"你想在全屏模式下运行么？", "设置全屏模式",MB_YESNO|MB_ICONQUESTION)==IDNO)
{
    fullscreen=FALSE; // FALSE为窗口模式
}
```

接着创建OpenGL窗口。CreateGLWindow函数的参数依次为标题、宽度、高度、色彩深度，以及全屏标志。就这么简单！我很欣赏这段代码的简洁。如果未能创建成功，函数返回FALSE。程序立即退出。

```
// 创建OpenGL窗口
if (!CreateGLWindow("NeHe's OpenGL程序框架",640,480,16,fullscreen))
{
    return 0; // 失败退出
}
```

下面是循环的开始。只要done保持FALSE，循环一直进行。

```
while(!done) // 保持循环直到 done=TRUE
{
```

我们要做的第一件事是检查是否有消息在等待。使用PeekMessage()可以在不锁住我们的程序的前提下对消息进行检查。许多程序使用GetMessage()，也可以很好的工作。但使用GetMessage()，程序在收到paint消息或其他别的什么窗口消息之前不会做任何事。

```
if (PeekMessage(&msg,NULL,0,0,PM_REMOVE)) // 有消息在等待吗?
{
```

下面的代码查看是否出现退出消息。如果当前的消息是由PostQuitMessage(0)引起的WM\_QUIT，done变量被设为TRUE，程序将退出。

```
if (msg.message==WM_QUIT) // 收到退出消息?
{
    done=TRUE; // 是，则done=TRUE
}
else // 不是，处理窗口消息
{
```

如果不是退出消息，我们翻译消息，然后发送消息，使得WndProc()或Windows能够处理他们。

```
    TranslateMessage(&msg); // 翻译消息
    DispatchMessage(&msg); // 发送消息
}
else // 如果没有消息
{
```

如果没有消息，绘制我们的OpenGL场景。代码的第一行查看窗口是否激活。如果按下ESC键，done变量被设为TRUE，程序将会退出。

```
// 绘制场景。监视ESC键和来自DrawGLScene()的退出消息
if (active) // 程序激活的么?
{
    if (keys[VK_ESCAPE]) // ESC 按下了么?
    {
        done=TRUE; // ESC 发出退出信号
    }
    else // 不是退出的时候，刷新屏幕
    {

```

如果程序是激活的且ESC没有按下，我们绘制场景并交换缓存(使用双缓存可以实现无闪烁的动画)。我们实际上在另一个看不见的"屏幕"上绘图。当我们交换缓存后，我们当前的屏幕被隐藏，现在看到的是刚才看不到的屏幕。这也是我们看不到场景绘制过程的原因。场景只是即时显示。

```
    DrawGLScene(); // 绘制场景
    SwapBuffers(hDC); // 交换缓存(双缓存)
}
}
```

下面的一点代码是最近新加的(05-01-00)。允许用户按下F1键在全屏模式和窗口模式间切换。

```
if (keys[VK_F1]) // F1键按下了么?
{
    keys[VK_F1]=FALSE; // 若是，使对应的Key数组中的值为 FALSE
    KillGLWindow(); // 销毁当前的窗口
    fullscreen=!fullscreen; // 切换 全屏 / 窗口 模式
    // 重建 OpenGL 窗口
}
```

```

        if (!CreateGLWindow("NeHe's OpenGL 程序框架",640,480,16,fullscreen))
        {
            return 0; // 如果窗口未能创建，程序退出
        }
    }
}

```

如果done变量不再是FALSE，程序退出。正常销毁OpenGL窗口，将所有的内存释放，退出程序。

```

// 关闭程序
KillGLWindow(); // 销毁窗口
return (msg.wParam); // 退出程序
}

```

在这一课中，我已试着尽量详细解释一切。每一步都与设置有关，并创建了一个全屏OpenGL程序。当您按下ESC键程序就会退出，并监视窗口是否激活。我花了整整2周时间来写代码，一周时间来改正BUG并讨论编程指南，2天（整整22小时来写HTML文件）。如果您有什么意见或建议请给我EMAIL。如果您认为有什么不对或可以改进，请告诉我。我想做最好的OpenGL教程并对您的反馈感兴趣。

#### 版权与使用声明：

我是个对学习和生活充满激情的普通男孩，在网络上我以DancingWind为昵称，我的联系方式是zhouwei02@mails.tsinghua.edu.cn，如果你有任何问题，都可以联系我。

#### 引子

网络是一个共享的资源，但我在自己的学习生涯中浪费大量的时间去搜索可用的资料，在现实生活中花费了大量的金钱和时间在书店中寻找资料，于是我给自己起了个昵称DancingWind，其意义是想风一样从各个知识的站点中吸取成长的养料。在飘荡了多年之后，我决定把自己收集的资料整理为一个统一的资源库。



### 版权声明

所有DancingWind发表的内容，大多都来自共享的资源，所以我没有资格把它们据为己有，或声称自己为这些资源作出了一点贡献。故任何人都可以复制，修改，重新发表，甚至以自己的名义发表，我都不会追究，但你在做以上事情的时候必须保证内容的完整性，给后来的人一个完整的教程。最后，任何人不能以这些资料的任何部分，谋取任何形式的报酬。

### 发展计划

在国外，很多资料都是很多人花费几年的时间慢慢积累起来的。如果任何人有兴趣与别人共享你的知识，我很欢迎你与我联系，但你必须同意我上面的声明。

### 感谢

感谢我的母亲一直以来对我的支持和在生活上的照顾。

感谢我深爱的女友田芹，一直以来默默的在精神上和生活中对我的支持，她甚至把买衣服的钱都用来给我买书了，她真的是我见过的最好的女孩，希望我能带给她幸福。

第02课 >