OpenGL 入门教程

1.第一课:

说起编程作图，大概还有很多人想起 TC 的#include <graphics.h>吧？
但是各位是否想过，那些画面绚丽的 PC 游戏是如何编写出来的？就靠 TC 那可怜的 640*480 分辨率、16 色来做吗？显然是不行的。

本帖的目的是让大家放弃 TC 的老旧图形接口，让大家接触一些新事物。

OpenGL 作为当前主流的图形 API 之一，它在一些场合具有比 DirectX 更优越的特性。
1、与 C 语言紧密结合。
OpenGL 命令最初就是用 C 语言函数来进行描述的，对于学习过 C 语言的人来讲，OpenGL 是容易理解和学习的。如果你曾经接触过 TC 的 graphics.h，你会发现，使用 OpenGL 作图甚至比 TC 更加简单。
2、强大的可移植性。
微软的 Direct3D 虽然也是十分优秀的图形 API，但它只用于 Windows 系统（现在还要加上一个 XBOX 游戏机）。而 OpenGL 不仅用于 Windows，还可以用于 Unix/Linux 等其它系统，它甚至在大型计算机、各种专业计算机（如：医疗用显示设备）上都有应用。并且，OpenGL 的基本命令都做到了硬件无关，甚至是平台无关。
3、高性能的图形渲染。
OpenGL 是一个工业标准，它的技术紧跟时代，现今各个显卡厂家无一不对 OpenGL 提供强力支持，激烈的竞争中使得 OpenGL 性能一直领先。
总之，OpenGL 是一个很 NB 的图形软件接口。至于究竟有多 NB，去看看 DOOM3 和 QUAKE4 等专业游戏就知道了。
OpenGL 官方网站（英文）
http://www.opengl.org

下面我将对 Windows 下的 OpenGL 编程进行简单介绍。
学习 OpenGL 前的准备工作
第一步，选择一个编译环境
现在 Windows 系统的主流编译环境有 Visual Studio，Broland C++ Builder，Dev-C++等，它们都是支持 OpenGL 的。但这里我们选择 Visual Studio 2005 作为学习 OpenGL 的环境。
第二步，安装 GLUT 工具包
GLUT 不是 OpenGL 所必须的，但它会给我们的学习带来一定的方便，推荐安装。
Windows 环境下的 GLUT 下载地址：（大小约为 150k）
http://www.opengl.org/resources/libraries/glut/glutdlls37beta.zip
无法从以上地址下载的话请使用下面的连接:
http://upload.programfan.com/upfile/200607311626279.zip
Windows 环境下安装 GLUT 的步骤：
1、将下载的压缩包解开，将得到 5 个文件
2、在“我的电脑”中搜索“gl.h”，并找到其所在文件夹（如果是 VisualStudio2005，则应该是其安装目录下面的“VC\PlatformSDK\include\gl 文件夹”）。把解压得到的 glut.h 放到这个文件夹。
3、把解压得到的 glut.lib 和 glut32.lib 放到静态函数库所在文件夹（如果是 VisualStudio2005，则应该是其安装目录下面的“VC\lib”文件夹）。

4、把解压得到的 glut.dll 和 glut32.dll 放到操作系统目录下面的 system32 文件夹内。（典型的位置为：C:\Windows\System32）

第三步，建立一个 OpenGL 工程

这里以 VisualStudio2005 为例。

选择 File->New->Project，然后选择 Win32 Console Application，选择一个名字，然后按 OK。

在谈出的对话框左边点 Application Settings，找到 Empty project 并勾上，选择 Finish。

然后向该工程添加一个代码文件，取名为“OpenGL.c”，注意用.c 来作为文件结尾。

搞定了，就跟平时的工程没什么两样的。

第一个 OpenGL 程序

一个简单的 OpenGL 程序如下：（注意，如果需要编译并运行，需要正确安装 GLUT，安装方法如上所述）

```
#include <GL/glut.h>

void myDisplay(void)
{
    glClear(GL_COLOR_BUFFER_BIT);
    glRectf(-0.5f, -0.5f, 0.5f, 0.5f);
    glFlush();
}

int main(int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(400, 400);
    glutCreateWindow("第一个 OpenGL 程序");
    glutDisplayFunc(&myDisplay);
    glutMainLoop();
    return 0;
}
```

该程序的作用是在一个黑色的窗口中央画一个白色的矩形。下面对各行语句进行说明。

怎么样？代码还不算长吧？

首先，需要包含头文件#include <GL/glut.h>，这是 GLUT 的头文件。

本来 OpenGL 程序一般还要包含<GL/gl.h>和<GL/glu.h>，但 GLUT 的头文件中已经自动将这两个文件包含了，不必再次包含。

然后看 main 函数。

int main(int argc, char *argv[])，这个是带命令行参数的 main 函数，各位应该见过吧？没见过的同志们请多翻翻书，等弄明白了再往下看。

注意 main 函数中的各语句，除了最后的 return 之外，其余全部以 glut 开头。这种以 glut 开头的函数都是

GLUT 工具包所提供的函数，下面对用到的几个函数进行介绍。

1、glutInit，对 GLUT 进行初始化，这个函数必须在其它的 GLUT 使用之前调用一次。其格式比较死板，一般照抄这句 glutInit(&argc, argv)就可以了。

2、glutInitDisplayMode，设置显示方式，其中 GLUT_RGB 表示使用 RGB 颜色，与之对应的还有 GLUT_INDEX（表示使用索引颜色）。GLUT_SINGLE 表示使用单缓冲，与之对应的还有 GLUT_DOUBLE（使用双缓冲）。更多信息，请自己 Google。当然以后的教程也会有一些讲解。

3、glutInitWindowPosition，这个简单，设置窗口在屏幕中的位置。

4、glutInitWindowSize，这个也简单，设置窗口的大小。

5、glutCreateWindow，根据前面设置的信息创建窗口。参数将被作为窗口的标题。注意：窗口被创建后，并不立即显示到屏幕上。需要调用 glutMainLoop 才能看到窗口。

6、glutDisplayFunc，设置一个函数，当需要进行画图时，这个函数就会被调用。（这个说法不够准确，但准确的说法可能初学者不太好理解，暂时这样说吧）。

7、glutMainLoop，进行一个消息循环。（这个可能初学者也不太明白，现在只需要知道这个函数可以显示窗口，并且等待窗口关闭后才会返回，这就足够了。）

在 glutDisplayFunc 函数中，我们设置了“当需要画图时，请调用 myDisplay 函数”。于是 myDisplay 函数就用来画图。观察 myDisplay 中的三个函数调用，发现它们都以 gl 开头。这种以 gl 开头的函数都是 OpenGL 的标准函数，下面对用到的函数进行介绍。

1、glClear，清除。GL_COLOR_BUFFER_BIT 表示清除颜色，glClear 函数还可以清除其它的东西，但这里不作介绍。

2、glRectf，画一个矩形。四个参数分别表示了位于对角线上的两个点的横、纵坐标。

3、glFlush，保证前面的 OpenGL 命令立即执行（而不是让它们在缓冲区中等待）。其作用跟 fflush(stdout)类似。

2.第二课：

本次课程所要讲的是绘制简单的几何图形，在实际绘制之前，让我们先熟悉一些概念。

一、点、直线和多边形

我们知道数学（具体的说，是几何学）中有点、直线和多边形的概念，但这些概念在计算机中会有所不同。数学上的点，只有位置，没有大小。但在计算机中，无论计算精度如何提高，始终不能表示一个无穷小的点。另一方面，无论图形输出设备（例如，显示器）如何精确，始终不能输出一个无穷小的点。一般情况下，OpenGL 中的点将被画成单个的像素（像素的概念，请自己搜索之~），虽然它可能足够小，但并不会是无穷小。同一像素上，OpenGL 可以绘制许多坐标只有稍微不同的点，但该像素的具体颜色将取决于 OpenGL 的实现。当然，过度的注意细节就是钻牛角尖，我们大可不必花费过多的精力去研究“多个点如何画到同一像素上”。

同样的，数学上的直线没有宽度，但 OpenGL 的直线则是有宽度的。同时，OpenGL 的直线必须是有限长度，而不是像数学概念那样是无限的。可以认为，OpenGL 的“直线”概念与数学上的“线段”接近，它可以由两个端点来确定。

多边形是由多条线段首尾相连而形成的闭合区域。OpenGL 规定，一个多边形必须是一个“凸多边形”（其定义为：多边形内任意两点所确定的线段都在多边形内，由此也可以推导出，凸多边形不能是空心的）。多边形可以由其边的端点（这里可称为顶点）来确定。（注意：如果使用的多边形不是凸多边形，则最后输出的效果是未定义的——OpenGL 为了效率，放宽了检查，这可能导致显示错误。要避免这个错误，尽量使用三角形，因为三角形都是凸多边形）

可以想象，通过点、直线和多边形，就可以组合成各种几何图形。甚至于，你可以把一段弧看成是很多短的直线段相连，这些直线段足够短，以至于其长度小于一个像素的宽度。这样一来弧和圆也可以表示出来了。通过位于不同平面的相连的小多边形，我们还可以组成一个“曲面”。

二、在 OpenGL 中指定顶点

由以上的讨论可以知道，“点”是一切的基础。

如何指定一个点呢？OpenGL 提供了一系列函数。它们都以 glVertex 开头，后面跟一个数字和 1~2 个字母。例如：

glVertex2d

glVertex2f

glVertex3f

glVertex3fv

等等。

数字表示参数的个数，2 表示有两个参数，3 表示三个，4 表示四个（我知道有点罗嗦~）。

字母表示参数的类型，s 表示 16 位整数（OpenGL 中将这个类型定义为 GLshort），

i 表示 32 位整数（OpenGL 中将这个类型定义为 GLint 和 GLsizei），

f 表示 32 位浮点数（OpenGL 中将这个类型定义为 GLfloat 和 GLclampf），

d 表示 64 位浮点数（OpenGL 中将这个类型定义为 GLdouble 和 GLclampd）。

v 表示传递的几个参数将使用指针的方式，见下面的例子。

这些函数除了参数的类型和个数不同以外，功能是相同的。例如，以下五个代码段的功能是等效的：

（一）glVertex2i(1, 3);

（二）glVertex2f(1.0f, 3.0f);

（三）glVertex3f(1.0f, 3.0f, 0.0f);

（四）glVertex4f(1.0f, 3.0f, 0.0f, 1.0f);

（五）GLfloat VertexArr3[] = {1.0f, 3.0f, 0.0f};

```
glVertex3fv(VertexArr3);
```

以后我们将用 glVertex*来表示这一系列函数。

注意：OpenGL 的很多函数都是采用这样的形式，一个相同的前缀再加上参数说明标记，这一点会随着学习的深入而有更多的体会。

三、开始绘制

假设现在我已经指定了若干顶点，那么 OpenGL 是如何知道我想拿这些顶点来干什么呢？是一个一个的画出来，还是连成线？或者构成一个多边形？或者做其它什么事情？

为了解决这一问题，OpenGL 要求：指定顶点的命令必须包含在 glBegin 函数之后，glEnd 函数之前（否则指定的顶点将被忽略）。并由 glBegin 来指明如何使用这些点。

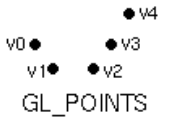
例如我写：

```
glBegin(GL_POINTS);
  glVertex2f(0.0f, 0.0f);
  glVertex2f(0.5f, 0.0f);
glEnd();
```

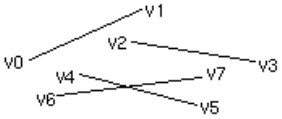
则这两个点将分别被画出来。如果将 GL_POINTS 替换成 GL_LINES，则两个点将被认为是直线的两个端点，OpenGL 将会画出一条直线。

我们还可以指定更多的顶点，然后画出更复杂的图形。

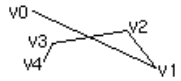
另一方面，glBegin 支持的方式除了 GL_POINTS 和 GL_LINES，还有 GL_LINE_STRIP，GL_LINE_LOOP，GL_TRIANGLES，GL_TRIANGLE_STRIP，GL_TRIANGLE_FAN 等，每种方式的大致效果见下图：



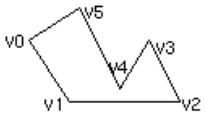
GL_POINTS



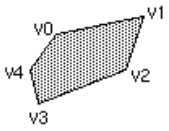
GL_LINES



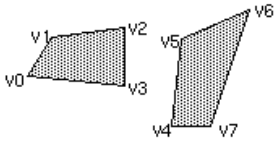
GL_LINE_STRIP



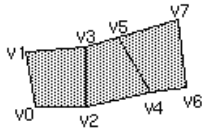
GL_LINE_LOOP



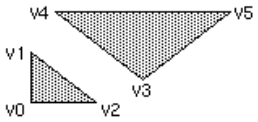
GL_POLYGON



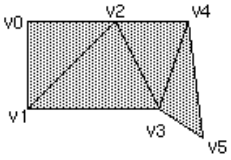
GL_QUADS



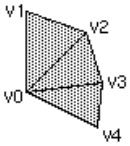
GL_QUAD_STRIP



GL_TRIANGLES



GL_TRIANGLE_STRIP



GL_TRIANGLE_FAN

声明：该图片来自 www.opengl.org，该图片是《OpenGL 编程指南》一书的附图，由于该书的旧版（第一版，1994 年）已经流传于网络，我希望没有触及到版权问题。

我并不准备在 glBegin 的各种方式上大作文章。大家可以自己尝试改变 glBegin 的方式和顶点的位置，生成一些有趣的图案。

程序代码：

```
void myDisplay(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin( /* 在这里填上你所希望的模式 */ );
    /* 在这里使用 glVertex*系列函数 */
    /* 指定你所希望的顶点位置 */
  glEnd();
  glFlush();
}
```

把这段代码改成你喜欢的样子，然后用它替换第一课中的 myDisplay 函数，编译后即可运行。

两个例子

例一、画一个圆

```
/*
正四边形，正五边形，正六边形，……，直到正 n 边形，当 n 越大时，这个图形就越接近圆
当 n 大到一定程度后，人眼将无法把它跟真正的圆相区别
这时我们已经成功的画出了一个“圆”
（注：画圆的方法很多，这里使用的是比较简单，但效率较低的一种）
```

试修改下面的 const int n 的值，观察当 n=3,4,5,8,10,15,20,30,50 等不同数值时输出的变化情况
将 GL_POLYGON 改为 GL_LINE_LOOP、GL_POINTS 等其它方式，观察输出的变化情况
*/

```
#include <math.h>
const int n = 20;
const GLfloat R = 0.5f;
const GLfloat Pi = 3.1415926536f;
void myDisplay(void)
{
   int i;
   glClear(GL_COLOR_BUFFER_BIT);
   glBegin(GL_POLYGON);
   for(i=0; i<n; ++i)
      glVertex2f(R*cos(2*Pi/n*i), R*sin(2*Pi/n*i));
   glEnd();
   glFlush();
}
```

例二、画一个五角星

/*
设五角星的五个顶点分布位置关系如下：

```
   A
E     B


  D  C
```

首先，根据余弦定理列方程，计算五角星的中心到顶点的距离 a
（假设五角星对应正五边形的边长为.0）

```
a = 1 / (2-2*cos(72*Pi/180));
```

然后，根据正弦和余弦的定义，计算 B 的 x 坐标 bx 和 y 坐标 by，以及 C 的 y 坐标
（假设五角星的中心在坐标原点）

```
bx = a * cos(18 * Pi/180);
by = a * sin(18 * Pi/180);
cy = -a * cos(18 * Pi/180);
```

五个点的坐标就可以通过以上四个量和一些常数简单的表示出来
*/

```
#include <math.h>
const GLfloat Pi = 3.1415926536f;
void myDisplay(void)
{
   GLfloat a = 1 / (2-2*cos(72*Pi/180));
   GLfloat bx = a * cos(18 * Pi/180);
   GLfloat by = a * sin(18 * Pi/180);
   GLfloat cy = -a * cos(18 * Pi/180);
   GLfloat
      PointA[2] = { 0, a },
```

```
    PointB[2] = { bx, by },
    PointC[2] = { 0.5, cy },
    PointD[2] = { -0.5, cy },
    PointE[2] = { -bx, by };

  glClear(GL_COLOR_BUFFER_BIT);
  // 按照 A->C->E->B->D->A 的顺序，可以一笔将五角星画出
  glBegin(GL_LINE_LOOP);
    glVertex2fv(PointA);
    glVertex2fv(PointC);
    glVertex2fv(PointE);
    glVertex2fv(PointB);
    glVertex2fv(PointD);
  glEnd();
  glFlush();
}
```

例三、画出正弦函数的图形

```
/*
由于 OpenGL 默认坐标值只能从-1 到 1，（可以修改，但方法留到以后讲）
所以我们设置一个因子 factor，把所有的坐标值等比例缩小，
这样就可以画出更多个正弦周期
试修改 factor 的值，观察变化情况
*/
#include <math.h>
const GLfloat factor = 0.1f;
void myDisplay(void)
{
  GLfloat x;
  glClear(GL_COLOR_BUFFER_BIT);
  glBegin(GL_LINES);
    glVertex2f(-1.0f, 0.0f);
    glVertex2f(1.0f, 0.0f);       // 以上两个点可以画 x 轴
    glVertex2f(0.0f, -1.0f);
    glVertex2f(0.0f, 1.0f);       // 以上两个点可以画 y 轴
  glEnd();
  glBegin(GL_LINE_STRIP);
  for(x=-1.0f/factor; x<1.0f/factor; x+=0.01f)
  {
    glVertex2f(x*factor, sin(x)*factor);
  }
  glEnd();
  glFlush();
}
```

小结

本课讲述了点、直线和多边形的概念，以及如何使用 OpenGL 来描述点，并使用点来描述几何图形。
大家可以发挥自己的想象，画出各种几何图形，当然，也可以用 GL_LINE_STRIP 把很多位置相近的点连接起来，构成函数图象。如果有兴趣，也可以去找一些图象比较美观的函数，自己动手，用 OpenGL 把它画出来。

3.第三课：

1、关于点

点的大小默认为 1 个像素，但也可以改变之。改变的命令为 glPointSize，其函数原型如下：

```
void glPointSize(GLfloat size);
```

size 必须大于 0.0f，默认值为 1.0f，单位为“像素”。
注意：对于具体的 OpenGL 实现，点的大小都有个限度的，如果设置的 size 超过最大值，则设置可能会有问题。
例子：

```
void myDisplay(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glPointSize(5.0f);
   glBegin(GL_POINTS);
      glVertex2f(0.0f, 0.0f);
      glVertex2f(0.5f, 0.5f);
   glEnd();
   glFlush();
}
```

2、关于直线

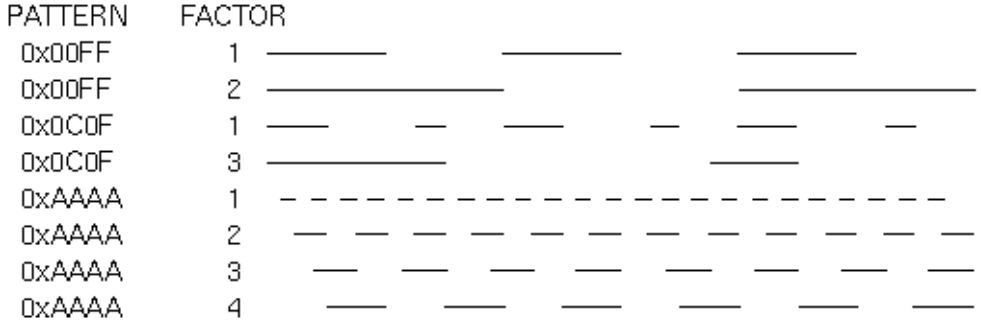
（1）直线可以指定宽度：

```
void glLineWidth(GLfloat width);
```

其用法跟 glPointSize 类似。
（2）画虚线。
首先，使用 glEnable(GL_LINE_STIPPLE);来启动虚线模式（使用 glDisable(GL_LINE_STIPPLE)可以关闭之）。然后，使用 glLineStipple 来设置虚线的样式。

```
void glLineStipple(GLint factor, GLushort pattern);
```

pattern 是由 1 和 0 组成的长度为 16 的序列，从最低位开始看，如果为 1，则直线上接下来应该画的 factor 个点将被画为实的；如果为 0，则直线上接下来应该画的 factor 个点将被画为虚的。
以下是一些例子：



声明：该图片来自 www.opengl.org，该图片是《OpenGL 编程指南》一书的附图，由于该书的旧版（第一版，

1994 年）已经流传于网络，我希望没有触及到版权问题。

示例代码：

```
void myDisplay(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glEnable(GL_LINE_STIPPLE);
  glLineStipple(2, 0x0F0F);
  glLineWidth(10.0f);
  glBegin(GL_LINES);
    glVertex2f(0.0f, 0.0f);
    glVertex2f(0.5f, 0.5f);
  glEnd();
  glFlush();
}
```

3、关于多边形

多边形的内容较多，我们将讲述以下四个方面。

（1）多边形的两面以及绘制方式。

虽然我们目前还没有真正的使用三维坐标来画图，但是建立一些三维的概念还是必要的。

从三维的角度来看，一个多边形具有两个面。每一个面都可以设置不同的绘制方式：填充、只绘制边缘轮廓线、只绘制顶点，其中“填充”是默认的方式。可以为两个面分别设置不同的方式。

```
glPolygonMode(GL_FRONT, GL_FILL);          // 设置正面为填充方式
glPolygonMode(GL_BACK, GL_LINE);           // 设置反面为边缘绘制方式
glPolygonMode(GL_FRONT_AND_BACK, GL_POINT); // 设置两面均为顶点绘制方式
```

（2）反转

一般约定为“顶点以逆时针顺序出现在屏幕上的面”为“正面”，另一个面即成为“反面”。生活中常见的物体表面，通常都可以用这样的“正面”和“反面”，“合理的”被表现出来（请找一个比较透明的矿泉水瓶子，在正对你的一面沿逆时针画一个圆，并标明画的方向，然后将背面转为正面，画一个类似的圆，体会一下“正面”和“反面”。你会发现正对你的方向，瓶的外侧是正面，而背对你的方向，瓶的内侧才是正面。正对你的内侧和背对你的外侧则是反面。这样一来，同样属于“瓶的外侧”这个表面，但某些地方算是正面，某些地方却算是反面了）。

但也有一些表面比较特殊。例如“麦比乌斯带”（请自己 Google 一下），可以全部使用“正面”或全部使用“背面”来表示。

可以通过 glFrontFace 函数来交换“正面”和“反面”的概念。

```
glFrontFace(GL_CCW); // 设置 CCW 方向为“正面”，CCW 即 CounterClockWise，逆时针
glFrontFace(GL_CW);  // 设置 CW 方向为“正面”，CW 即 ClockWise，顺时针
```

下面是一个示例程序，请用它替换第一课中的 myDisplay 函数，并将 glFrontFace(GL_CCW)修改为 glFrontFace(GL_CW)，并观察结果的变化。

```
void myDisplay(void)
{
  glClear(GL_COLOR_BUFFER_BIT);
  glPolygonMode(GL_FRONT, GL_FILL); // 设置正面为填充模式
  glPolygonMode(GL_BACK, GL_LINE);  // 设置反面为线形模式
  glFrontFace(GL_CCW);              // 设置逆时针方向为正面
  glBegin(GL_POLYGON);              // 按逆时针绘制一个正方形，在左下方
```

```
        glVertex2f(-0.5f, -0.5f);
        glVertex2f(0.0f, -0.5f);
        glVertex2f(0.0f, 0.0f);
        glVertex2f(-0.5f, 0.0f);
    glEnd();
    glBegin(GL_POLYGON);            // 按顺时针绘制一个正方形，在右上方
        glVertex2f(0.0f, 0.0f);
        glVertex2f(0.0f, 0.5f);
        glVertex2f(0.5f, 0.5f);
        glVertex2f(0.5f, 0.0f);
    glEnd();
    glFlush();
}
```

（3）剔除多边形表面

在三维空间中，一个多边形虽然有两个面，但我们无法看见背面的那些多边形，而一些多边形虽然是正面的，但被其他多边形所遮挡。如果将无法看见的多边形和可见的多边形同等对待，无疑会降低我们处理图形的效率。在这种时候，可以将不必要的面剔除。

首先，使用 glEnable(GL_CULL_FACE);来启动剔除功能（使用 glDisable(GL_CULL_FACE)可以关闭之）

然后，使用 glCullFace 来进行剔除。

glCullFace 的参数可以是 GL_FRONT，GL_BACK 或者 GL_FRONT_AND_BACK，分别表示剔除正面、剔除反面、剔除正反两面的多边形。

注意：剔除功能只影响多边形，而对点和直线无影响。例如，使用 glCullFace(GL_FRONT_AND_BACK)后，所有的多边形都将被剔除，所以看见的就只有点和直线。

（4）镂空多边形

直线可以被画成虚线，而多边形则可以进行镂空。

首先，使用 glEnable(GL_POLYGON_STIPPLE);来启动镂空模式（使用 glDisable(GL_POLYGON_STIPPLE)可以关闭之）。

然后，使用 glPolygonStipple 来设置镂空的样式。

void glPolygonStipple(const GLubyte *mask);

其中的参数 mask 指向一个长度为 128 字节的空间，它表示了一个 32*32 的矩形应该如何镂空。其中：第一个字节表示了最左下方的从左到右（也可以是从右到左，这个可以修改）8 个像素是否镂空（1 表示不镂空，显示该像素；0 表示镂空，显示其后面的颜色），最后一个字节表示了最右上方的 8 个像素是否镂空。

但是，如果我们直接定义这个 mask 数组，像这样：

```
static GLubyte Mask[128] =
{
    0x00, 0x00, 0x00, 0x00,   // 这是最下面的一行
    0x00, 0x00, 0x00, 0x00,
    0x03, 0x80, 0x01, 0xC0,   // 麻
    0x06, 0xC0, 0x03, 0x60,   // 烦
    0x04, 0x60, 0x06, 0x20,   // 的
    0x04, 0x30, 0x0C, 0x20,   // 初
    0x04, 0x18, 0x18, 0x20,   // 始
    0x04, 0x0C, 0x30, 0x20,   // 化
```

```
    0x04, 0x06, 0x60, 0x20,   // ，
    0x44, 0x03, 0xC0, 0x22,   // 不
    0x44, 0x01, 0x80, 0x22,   // 建
    0x44, 0x01, 0x80, 0x22,   // 议
    0x44, 0x01, 0x80, 0x22,   // 使
    0x44, 0x01, 0x80, 0x22,   // 用
    0x44, 0x01, 0x80, 0x22,
    0x44, 0x01, 0x80, 0x22,
    0x66, 0x01, 0x80, 0x66,
    0x33, 0x01, 0x80, 0xCC,
    0x19, 0x81, 0x81, 0x98,
    0x0C, 0xC1, 0x83, 0x30,
    0x07, 0xE1, 0x87, 0xE0,
    0x03, 0x3F, 0xFC, 0xC0,
    0x03, 0x31, 0x8C, 0xC0,
    0x03, 0x3F, 0xFC, 0xC0,
    0x06, 0x64, 0x26, 0x60,
    0x0C, 0xCC, 0x33, 0x30,
    0x18, 0xCC, 0x33, 0x18,
    0x10, 0xC4, 0x23, 0x08,
    0x10, 0x63, 0xC6, 0x08,
    0x10, 0x30, 0x0C, 0x08,
    0x10, 0x18, 0x18, 0x08,
    0x10, 0x00, 0x00, 0x08   // 这是最上面的一行
};
```

这样一堆数据非常缺乏直观性，我们需要很费劲的去分析，才会发现它表示的竟然是一只苍蝇。

如果将这样的数据保存成图片，并用专门的工具进行编辑，显然会方便很多。下面介绍如何做到这一点。

首先，用 Windows 自带的画笔程序新建一副图片，取名为 mask.bmp，注意保存时，应该选择“单色位图”。

在“图象”->“属性”对话框中，设置图片的高度和宽度均为 32。

用放大镜观察图片，并编辑之。黑色对应二进制零（镂空），白色对应二进制一（不镂空），编辑完毕后保存。

然后，就可以使用以下代码来获得这个 Mask 数组了。

```
static GLubyte Mask[128];
FILE *fp;
fp = fopen("mask.bmp", "rb");
if( !fp )
    exit(0);
// 移动文件指针到这个位置，使得再读 sizeof(Mask)个字节就会遇到文件结束
// 注意-(int)sizeof(Mask)虽然不是什么好的写法，但这里它确实是正确有效的
// 如果直接写-sizeof(Mask)的话，因为 sizeof 取得的是一个无符号数，取负号会有问题
if( fseek(fp, -(int)sizeof(Mask), SEEK_END) )
    exit(0);
// 读取 sizeof(Mask)个字节到 Mask
if( !fread(Mask, sizeof(Mask), 1, fp) )
```

```
    exit(0);
fclose(fp);
```

好的，现在请自己编辑一个图片作为 mask，并用上述方法取得 Mask 数组，运行后观察效果。

说明：绘制虚线时可以设置 factor 因子，但多边形的镂空无法设置 factor 因子。请用鼠标改变窗口的大小，观察镂空效果的变化情况。

```
#include <stdio.h>
#include <stdlib.h>
void myDisplay(void)
{
    static GLubyte Mask[128];
    FILE *fp;
    fp = fopen("mask.bmp", "rb");
    if( !fp )
        exit(0);
    if( fseek(fp, -(int)sizeof(Mask), SEEK_END) )
        exit(0);
    if( !fread(Mask, sizeof(Mask), 1, fp) )
        exit(0);
    fclose(fp);
    glClear(GL_COLOR_BUFFER_BIT);
    glEnable(GL_POLYGON_STIPPLE);
    glPolygonStipple(Mask);
    glRectf(-0.5f, -0.5f, 0.0f, 0.0f);  // 在左下方绘制一个有镂空效果的正方形
    glDisable(GL_POLYGON_STIPPLE);
    glRectf(0.0f, 0.0f, 0.5f, 0.5f);    // 在右上方绘制一个无镂空效果的正方形
    glFlush();
}
```

小结

本课学习了绘制几何图形的一些细节。

点可以设置大小。

直线可以设置宽度；可以将直线画成虚线。

多边形的两个面的绘制方法可以分别设置；在三维空间中，不可见的多边形可以被剔除；可以将填充多边形绘制成镂空的样式。

了解这些细节会使我们在一些图象绘制中更加得心应手。

另外，把一些数据写到程序之外的文件中，并用专门的工具编辑之，有时可以显得更方便。

4.第四课：

OpenGL 支持两种颜色模式：一种是 RGBA，一种是颜色索引模式。

无论哪种颜色模式，计算机都必须为每一个像素保存一些数据。不同的是，RGBA 模式中，数据直接就代表了颜色；而颜色索引模式中，数据代表的是一个索引，要得到真正的颜色，还必须去查索引表。

1. RGBA 颜色

RGBA 模式中，每一个像素会保存以下数据：R 值（红色分量）、G 值（绿色分量）、B 值（蓝色分量）和 A 值（alpha 分量）。其中红、绿、蓝三种颜色相组合，就可以得到我们所需要的各种颜色，而 alpha 不直接影响颜色，它将留待以后介绍。

在 RGBA 模式下选择颜色是十分简单的事情，只需要一个函数就可以搞定。

glColor*系列函数可以用于设置颜色，其中三个参数的版本可以指定 R、G、B 的值，而 A 值采用默认；四个参数的版本可以分别指定 R、G、B、A 的值。例如：

void glColor3f(GLfloat red, GLfloat green, GLfloat blue);

void glColor4f(GLfloat red, GLfloat green, GLfloat blue, GLfloat alpha);

（还记得吗？3f 表示有三个浮点参数~请看第二课中关于 glVertex*函数的叙述。）

将浮点数作为参数，其中 0.0 表示不使用该种颜色，而 1.0 表示将该种颜色用到最多。例如：

glColor3f(1.0f, 0.0f, 0.0f); 表示不使用绿、蓝色，而将红色使用最多，于是得到最纯净的红色。

glColor3f(0.0f, 1.0f, 1.0f); 表示使用绿、蓝色到最多，而不使用红色。混合的效果就是浅蓝色。

glColor3f(0.5f, 0.5f, 0.5f); 表示各种颜色使用一半，效果为灰色。

注意：浮点数可以精确到小数点后若干位，这并不表示计算机就可以显示如此多种颜色。实际上，计算机可以显示的颜色种数将由硬件决定。如果 OpenGL 找不到精确的颜色，会进行类似“四舍五入”的处理。

大家可以通过改变下面代码中 glColor3f 的参数值，绘制不同颜色的矩形。

```
void myDisplay(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glColor3f(0.0f, 1.0f, 1.0f);
   glRectf(-0.5f, -0.5f, 0.5f, 0.5f);
   glFlush();
}
```

注意：glColor 系列函数，在参数类型不同时，表示“最大”颜色的值也不同。

采用 f 和 d 做后缀的函数，以 1.0 表示最大的使用。

采用 b 做后缀的函数，以 127 表示最大的使用。

采用 ub 做后缀的函数，以 255 表示最大的使用。

采用 s 做后缀的函数，以 32767 表示最大的使用。

采用 us 做后缀的函数，以 65535 表示最大的使用。

这些规则看似麻烦，但熟悉后实际使用中不会有什么障碍。

2、索引颜色

在索引颜色模式中，OpenGL 需要一个颜色表。这个表就相当于画家的调色板：虽然可以调出很多种颜色，但同时存在于调色板上的颜色种数将不会超过调色板的格数。试将颜色表的每一项想象成调色板上的一个格子：它保存了一种颜色。

在使用索引颜色模式画图时，我说“我把第 i 种颜色设置为某某”，其实就相当于将调色板的第 i 格调为某某颜色。“我需要第 k 种颜色来画图”，那么就用画笔去蘸一下第 k 格调色板。

颜色表的大小是很有限的，一般在 256~4096 之间，且总是 2 的整数次幂。在使用索引颜色方式进行绘图时，总是先设置颜色表，然后选择颜色。

2.1、选择颜色

使用 glIndex*系列函数可以在颜色表中选择颜色。其中最常用的可能是 glIndexi，它的参数是一个整形。

void glIndexi(GLint c);

是的，这的确很简单。

2.2、设置颜色表

OpenGL 并直接没有提供设置颜色表的方法，因此设置颜色表需要使用操作系统的支持。我们所用的 Windows 和其他大多数图形操作系统都具有这个功能，但所使用的函数却不相同。正如我没有讲述如何自己写代码在 Windows 下建立一个窗口，这里我也不会讲述如何在 Windows 下设置颜色表。

GLUT 工具包提供了设置颜色表的函数 glutSetColor，但我测试始终有问题。现在为了让大家体验一下索引颜色，我向大家介绍另一个 OpenGL 工具包：aux。这个工具包是 VisualStudio 自带的，不必另外安装，但它已经过时，这里仅仅是体验一下，大家不必深入。

```
#include <windows.h>
#include <GL/gl.h>
#include <GL/glaux.h>

#pragma comment (lib, "opengl32.lib")
#pragma comment (lib, "glaux.lib")

#include <math.h>
const GLdouble Pi = 3.1415926536;
void myDisplay(void)
{
   int i;
   for(i=0; i<8; ++i)
      auxSetOneColor(i, (float)(i&0x04), (float)(i&0x02), (float)(i&0x01));
   glShadeModel(GL_FLAT);
   glClear(GL_COLOR_BUFFER_BIT);
   glBegin(GL_TRIANGLE_FAN);
   glVertex2f(0.0f, 0.0f);
   for(i=0; i<=8; ++i)
   {
      glIndexi(i);
      glVertex2f(cos(i*Pi/4), sin(i*Pi/4));
   }
   glEnd();
   glFlush();
}

int main(void)
{
   auxInitDisplayMode(AUX_SINGLE|AUX_INDEX);
   auxInitPosition(0, 0, 400, 400);
   auxInitWindow(L"");
   myDisplay();
   Sleep(10 * 1000);
   return 0;
```

}

其它部分大家都可以不管，只看 myDisplay 函数就可以了。首先，使用 auxSetOneColor 设置颜色表中的一格。循环八次就可以设置八格。

glShadeModel 等下再讲，这里不提。

然后在循环中用 glVertex 设置顶点，同时用 glIndexi 改变顶点代表的颜色。

最终得到的效果是八个相同形状、不同颜色的三角形。

索引颜色虽然讲得多了点。索引颜色的主要优势是占用空间小（每个像素不必单独保存自己的颜色，只用很少的二进制位就可以代表其颜色在颜色表中的位置），花费系统资源少，图形运算速度快，但它编程稍稍显得不是那么方便，并且画面效果也会比 RGB 颜色差一些。“星际争霸”可能代表了 256 色的颜色表的画面效果，虽然它在一台很烂的 PC 上也可以运行很流畅，但以目前的眼光来看，其画面效果就显得不足了。目前的 PC 机性能已经足够在各种场合下使用 RGB 颜色，因此 PC 程序开发中，使用索引颜色已经不是主流。当然，一些小型设备例如 GBA、手机等，索引颜色还是有它的用武之地。

3、指定清除屏幕用的颜色

我们写：glClear(GL_COLOR_BUFFER_BIT);意思是把屏幕上的颜色清空。

但实际上什么才叫“空”呢？在宇宙中，黑色代表了“空”；在一张白纸上，白色代表了“空”；在信封上，信封的颜色才是“空”。

OpenGL 用下面的函数来定义清楚屏幕后屏幕所拥有的颜色。

在 RGB 模式下，使用 glClearColor 来指定“空”的颜色，它需要四个参数，其参数的意义跟 glColor4f 相似。

在索引颜色模式下，使用 glClearIndex 来指定“空”的颜色所在的索引，它需要一个参数，其意义跟 glIndexi 相似。

```
void myDisplay(void)
{
   glClearColor(1.0f, 0.0f, 0.0f, 0.0f);
   glClear(GL_COLOR_BUFFER_BIT);
   glFlush();
}
```

呵，这个还真简单~

4、指定着色模型

OpenGL 允许为同一多边形的不同顶点指定不同的颜色。例如：

```
#include <math.h>
const GLdouble Pi = 3.1415926536;
void myDisplay(void)
{
   int i;
   // glShadeModel(GL_FLAT);
   glClear(GL_COLOR_BUFFER_BIT);
   glBegin(GL_TRIANGLE_FAN);
   glColor3f(1.0f, 1.0f, 1.0f);
   glVertex2f(0.0f, 0.0f);
   for(i=0; i<=8; ++i)
   {
      glColor3f(i&0x04, i&0x02, i&0x01);
```

```
        glVertex2f(cos(i*Pi/4), sin(i*Pi/4));
    }
    glEnd();
    glFlush();
}
```

在默认情况下，OpenGL 会计算两点顶点之间的其它点，并为它们填上“合适”的颜色，使相邻的点的颜色值都比较接近。如果使用的是 RGB 模式，看起来就具有渐变的效果。如果是使用颜色索引模式，则其相邻点的索引值是接近的，如果将颜色表中接近的项设置成接近的颜色，则看起来也是渐变的效果。但如果颜色表中接近的项颜色却差距很大，则看起来可能是很奇怪的效果。

使用 glShadeModel 函数可以关闭这种计算，如果顶点的颜色不同，则将顶点之间的其它点全部设置为与某一个点相同。（直线以后指定的点的颜色为准，而多边形将以任意顶点的颜色为准，由实现决定。）为了避免这个不确定性，尽量在多边形中使用同一种颜色。

glShadeModel 的使用方法：

```
glShadeModel(GL_SMOOTH);  // 平滑方式，这也是默认方式
glShadeModel(GL_FLAT);    // 单色方式
```

小结：

本课学习了如何设置颜色。其中 RGB 颜色方式是目前 PC 机上的常用方式。

可以设置 glClear 清除后屏幕所剩的颜色。

可以设置颜色填充方式：平滑方式或单色方式。

5. 第五课：

在前面绘制几何图形的时候，大家是否觉得我们绘图的范围太狭隘了呢？坐标只能从-1 到 1，还只能是 X 轴向右，Y 轴向上，Z 轴垂直屏幕。这些限制给我们的绘图带来了很多不便。

我们生活在一个三维的世界——如果要观察一个物体，我们可以：

1、从不同的位置去观察它。（视图变换）

2、移动或者旋转它，当然了，如果它只是计算机里面的物体，我们还可以放大或缩小它。（模型变换）

3、如果把物体画下来，我们可以选择：是否需要一种“近大远小”的透视效果。另外，我们可能只希望看到物体的一部分，而不是全部（剪裁）。（投影变换）

4、我们可能希望把整个看到的图形画下来，但它只占据纸张的一部分，而不是全部。（视口变换）

这些，都可以在 OpenGL 中实现。

OpenGL 变换实际上是通过矩阵乘法来实现。无论是移动、旋转还是缩放大小，都是通过在当前矩阵的基础上乘以一个新的矩阵来达到目的。关于矩阵的知识，这里不详细介绍，有兴趣的朋友可以看看线性代数（大学生的话多半应该学过的）。

OpenGL 可以在最底层直接操作矩阵，不过作为初学，这样做的意义并不大。这里就不做介绍了。

1、 模型变换和视图变换

从“相对移动”的观点来看，改变观察点的位置与方向和改变物体本身的位置与方向具有等效性。

在 OpenGL 中，实现这两种功能甚至使用的是同样的函数。

由于模型和视图的变换都通过矩阵运算来实现，在进行变换前，应先设置当前操作的矩阵为“模型视图矩阵”。设置的方法是以 GL_MODELVIEW 为参数调用 glMatrixMode 函数，像这样：

```
glMatrixMode(GL_MODELVIEW);
```

通常，我们需要在进行变换前把当前矩阵设置为单位矩阵。这也只需要一行代码：

```
glLoadIdentity();
```

然后，就可以进行模型变换和视图变换了。进行模型和视图变换，主要涉及到三个函数：

glTranslate*，把当前矩阵和一个表示移动物体的矩阵相乘。三个参数分别表示了在三个坐标上的位移值。

glRotate*，把当前矩阵和一个表示旋转物体的矩阵相乘。物体将绕着(0,0,0)到(x,y,z)的直线以逆时针旋转，参数 angle 表示旋转的角度。

glScale*，把当前矩阵和一个表示缩放物体的矩阵相乘。x,y,z 分别表示在该方向上的缩放比例。

注意我都是说“与 XX 相乘”，而不是直接说“这个函数就是旋转”或者“这个函数就是移动”，这是有原因的，马上就会讲到。

假设当前矩阵为单位矩阵，然后先乘以一个表示旋转的矩阵 R，再乘以一个表示移动的矩阵 T，最后得到的矩阵再乘上每一个顶点的坐标矩阵 v。所以，经过变换得到的顶点坐标就是((RT)v)。由于矩阵乘法的结合率，((RT)v) = (R(Tv))，换句话说，实际上是先进行移动，然后进行旋转。即：实际变换的顺序与代码中写的顺序是相反的。由于“先移动后旋转”和“先旋转后移动”得到的结果很可能不同，初学的时候需要特别注意这一点。

OpenGL 之所以这样设计，是为了得到更高的效率。但在绘制复杂的三维图形时，如果每次都去考虑如何把变换倒过来，也是很痛苦的事情。这里介绍另一种思路，可以让代码看起来更自然（写出的代码其实完全一样，只是考虑问题时用的方法不同了）。

让我们想象，坐标并不是固定不变的。旋转的时候，坐标系统随着物体旋转。移动的时候，坐标系统随着物体移动。如此一来，就不需要考虑代码的顺序反转的问题了。

以上都是针对改变物体的位置和方向来介绍的。如果要改变观察点的位置，除了配合使用 glRotate* 和 glTranslate*函数以外，还可以使用这个函数：gluLookAt。它的参数比较多，前三个参数表示了观察点的位置，中间三个参数表示了观察目标的位置，最后三个参数代表从(0,0,0)到(x,y,z)的直线，它表示了观察者认为的“上”方向。

2、投影变换

投影变换就是定义一个可视空间，可视空间以外的物体不会被绘制到屏幕上。（注意，从现在起，坐标可以不再是-1.0 到 1.0 了！）

OpenGL 支持两种类型的投影变换，即透视投影和正投影。投影也是使用矩阵来实现的。如果需要操作投影矩阵，需要以 GL_PROJECTION 为参数调用 glMatrixMode 函数。

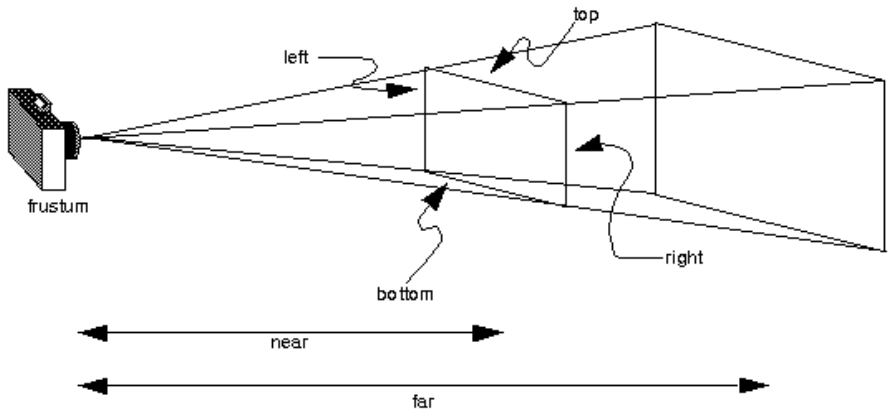
```
glMatrixMode(GL_PROJECTION);
```

通常，我们需要在进行变换前把当前矩阵设置为单位矩阵。

```
glLoadIdentity();
```

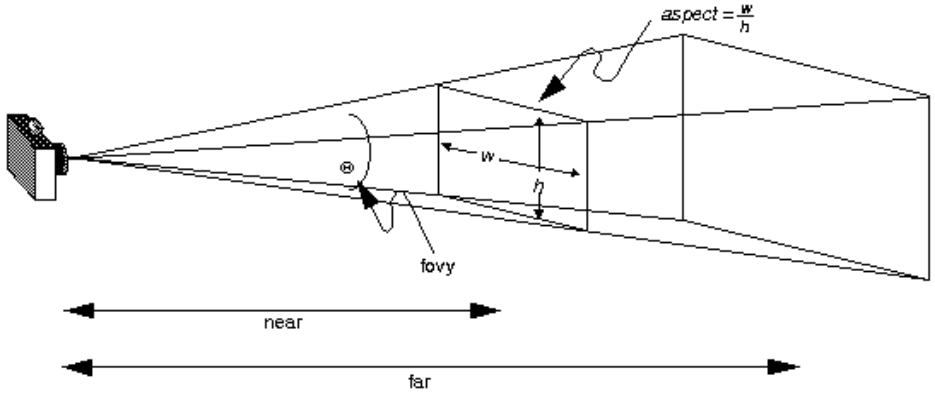
透视投影所产生的结果类似于照片，有近大远小的效果，比如在火车头内向前照一个铁轨的照片，两条铁轨似乎在远处相交了。

使用 glFrustum 函数可以将当前的可视空间设置为透视投影空间。其参数的意义如下图：



声明：该图片来自 www.opengl.org，该图片是《OpenGL 编程指南》一书的附图，由于该书的旧版（第一版，1994 年）已经流传于网络，我希望没有触及到版权问题。

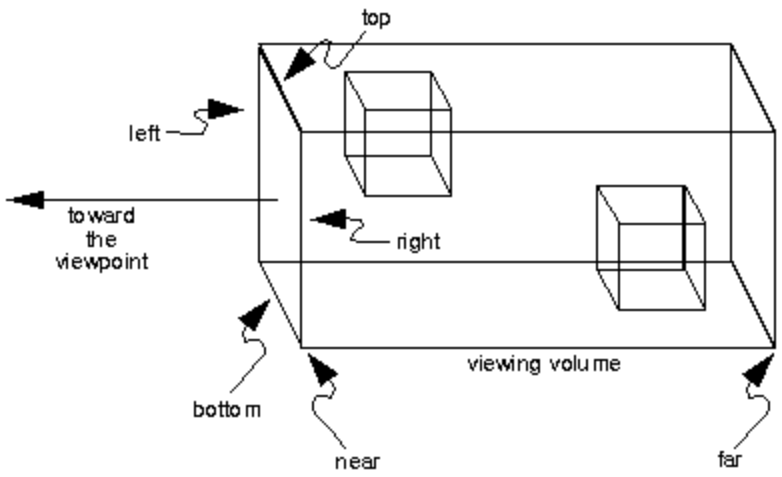
也可以使用更常用的 gluPerspective 函数。其参数的意义如下图：



声明：该图片来自 www.opengl.org，该图片是《OpenGL 编程指南》一书的附图，由于该书的旧版（第一版，1994 年）已经流传于网络，我希望没有触及到版权问题。

正投影相当于在无限远处观察得到的结果，它只是一种理想状态。但对于计算机来说，使用正投影有可能获得更好的运行速度。

使用 glOrtho 函数可以将当前的可视空间设置为正投影空间。其参数的意义如下图：

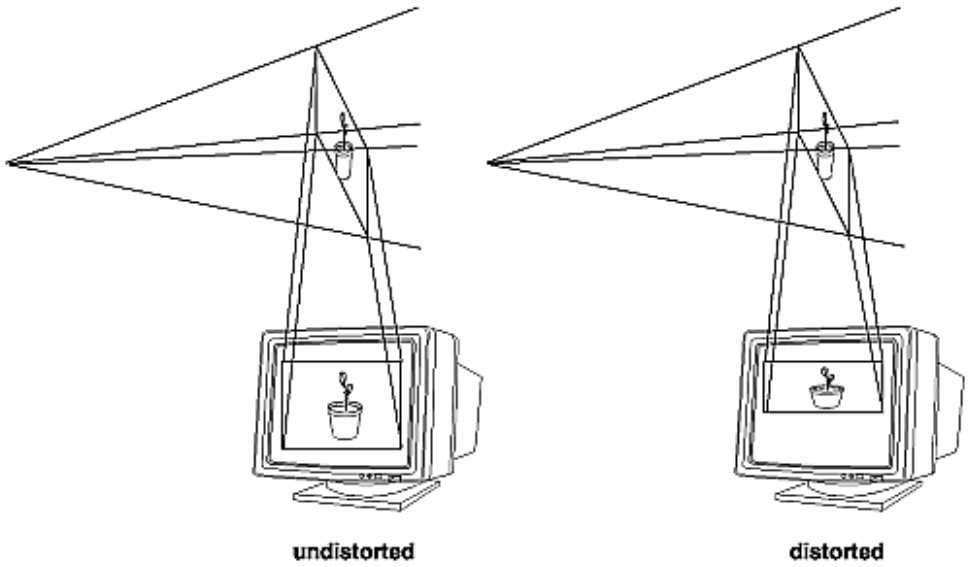


声明：该图片来自 www.opengl.org，该图片是《OpenGL 编程指南》一书的附图，由于该书的旧版（第一版，1994 年）已经流传于网络，我希望没有触及到版权问题。

如果绘制的图形空间本身就是二维的，可以使用 gluOrtho2D。他的使用类似于 glOrgho。

3、视口变换

当一切工作已经就绪，只需要把像素绘制到屏幕上了。这时候还剩最后一个问题：应该把像素绘制到窗口的哪个区域呢？通常情况下，默认是完整的填充整个窗口，但我们完全可以只填充一半。（即：把整个图象填充到一半的窗口内）



声明：该图片来自 www.opengl.org，该图片是《OpenGL 编程指南》一书的附图，由于该书的旧版（第一版，1994 年）已经流传于网络，我希望没有触及到版权问题。

使用 glViewport 来定义视口。其中前两个参数定义了视口的左下脚（0,0 表示最左下方），后两个参数分别是宽度和高度。

4、操作矩阵堆栈

介于是入门教程，先简单介绍一下堆栈。你可以把堆栈想象成一叠盘子。开始的时候一个盘子也没有，你可以一个一个往上放，也可以一个一个取下来。每次取下的，都是最后一次被放上去的盘子。通常，在计算机实现堆栈时，堆栈的容量是有限的，如果盘子过多，就会出错。当然，如果没有盘子了，再要求取一个盘子，也会出错。

我们在进行矩阵操作时，有可能需要先保存某个矩阵，过一段时间再恢复它。当我们需要保存时，调用 glPushMatrix 函数，它相当于把矩阵（相当于盘子）放到堆栈上。当需要恢复最近一次的保存时，调用 glPopMatrix 函数，它相当于把矩阵从堆栈上取下。OpenGL 规定堆栈的容量至少可以容纳 32 个矩阵，某些 OpenGL 实现中，堆栈的容量实际上超过了 32 个。因此不必过于担心矩阵的容量问题。

通常，用这种先保存后恢复的措施，比先变换再逆变换要更方便，更快速。

注意：模型视图矩阵和投影矩阵都有相应的堆栈。使用 glMatrixMode 来指定当前操作的究竟是模型视图矩阵还是投影矩阵。

5、综合举例

好了，视图变换的入门知识差不多就讲完了。但我们不能就这样结束。因为本次课程的内容实在

过于枯燥，如果分别举例，可能效果不佳。我只好综合的讲一个例子，算是给大家一个参考。至于实际的掌握，还要靠大家自己花功夫。闲话少说，现在进入正题。

我们要制作的是一个三维场景，包括了太阳、地球和月亮。假定一年有 12 个月，每个月 30 天。每年，地球绕着太阳转一圈。每个月，月亮围着地球转一圈。即一年有 360 天。现在给出日期的编号（0~359），要求绘制出太阳、地球、月亮的相对位置示意图。（这是为了编程方便才这样设计的。如果需要制作更现实的情况，那也只是一些数值处理而已，与 OpenGL 关系不大）

首先，让我们认定这三个天体都是球形，且他们的运动轨迹处于同一水平面，建立以下坐标系：太阳的中心为原点，天体轨迹所在的平面表示了 X 轴与 Y 轴决定的平面，且每年第一天，地球在 X 轴正方向上，月亮在地球的正 X 轴方向。

下一步是确立可视空间。注意：太阳的半径要比太阳到地球的距离短得多。如果我们直接使用天文观测得到的长度比例，则当整个窗口表示地球轨道大小时，太阳的大小将被忽略。因此，我们只能成倍的放大几个天体的半径，以适应我们观察的需要。（百度一下，得到太阳、地球、月亮的大致半径分别是：696000km，6378km，1738km。地球到太阳的距离约为 1.5 亿 km=150000000km，月亮到地球的距离约为 380000km。）

让我们假想一些数据，将三个天体的半径分别“修改”为：69600000（放大 100 倍），15945000（放大 2500 倍），4345000（放大 2500 倍）。将地球到月亮的距离“修改”为 38000000（放大 100 倍）。地球到太阳的距离保持不变。

为了让地球和月亮在离我们很近时，我们仍然不需要变换观察点和观察方向就可以观察它们，我们把观察点放在这个位置：(0, -200000000, 0)——因为地球轨道半径为 150000000，咱们就凑个整，取-200000000 就可以了。观察目标设置为原点（即太阳中心），选择 Z 轴正方向作为“上”方。当然我们还可以把观察点往“上”方移动一些，得到(0, -200000000, 200000000)，这样可以得到 45 度角的俯视效果。

为了得到透视效果，我们使用 gluPerspective 来设置可视空间。假定可视角为 60 度（如果调试时发现该角度不合适，可修改之。我在最后选择的数值是 75。），高宽比为 1.0。最近可视距离为 1.0，最远可视距离为 200000000*2=400000000。即：gluPerspective(60, 1, 1, 400000000);

现在我们来看看如何绘制这三个天体。

为了简单起见，我们把三个天体都想象成规则的球体。而我们所使用的 glut 实用工具中，正好就有一个绘制球体的现成函数：glutSolidSphere，这个函数在“原点”绘制出一个球体。由于坐标是可以通过 glTranslate*和 glRotate*两个函数进行随意变换的，所以我们就可以在任意位置绘制球体了。函数有三个参数：第一个参数表示球体的半径，后两个参数代表了“面”的数目，简单点说就是球体的精确程度，数值越大越精确，当然代价就是速度越缓慢。这里我们只是简单的设置后两个参数为 20。

太阳在坐标原点，所以不需要经过任何变换，直接绘制就可以了。

地球则要复杂一点，需要变换坐标。由于今年已经经过的天数已知为 day，则地球转过的角度为 day/一年的天数*360 度。前面已经假定每年都是 360 天，因此地球转过的角度恰好为 day。所以可以通过下面的代码来解决：

```
glRotatef(day, 0, 0, -1);
/* 注意地球公转是“自西向东”的，因此是饶着 Z 轴负方向进行逆时针旋转 */
glTranslatef(地球轨道半径, 0, 0);
glutSolidSphere(地球半径, 20, 20);
```

月亮是最复杂的。因为它不仅要绕地球转，还要随着地球绕太阳转。但如果我们选择地球作为参考，则月亮进行的运动就是一个简单的圆周运动了。如果我们先绘制地球，再绘制月亮，则只需要进行与地球类似的变换：

glRotatef(月亮旋转的角度, 0, 0, -1);

glTranslatef(月亮轨道半径, 0, 0);

glutSolidSphere(月亮半径, 20, 20);

但这个“月亮旋转的角度”，并不能简单的理解为 day/一个月的天数 30*360 度。因为我们在绘制地球时，这个坐标已经是旋转过的。现在的旋转是在以前的基础上进行旋转，因此还需要处理这个“差值”。我们可以写成：day/30*360 - day，即减去原来已经转过的角度。这只是一种简单的处理，当然也可以在绘制地球前用 glPushMatrix 保存矩阵，绘制地球后用 glPopMatrix 恢复矩阵。再设计一个跟地球位置无关的月亮位置公式，来绘制月亮。通常后一种方法比前一种要好，因为浮点的运算是不精确的，即是说我们计算地球本身的位置就是不精确的。拿这个不精确的数去计算月亮的位置，会导致“不精确”的成分累积，过多的“不精确”会造成错误。我们这个小程序没有去考虑这个，但并不是说这个问题不重要。

还有一个需要注意的细节：OpenGL 把三维坐标中的物体绘制到二维屏幕，绘制的顺序是按照代码的顺序来进行的。因此后绘制的物体会遮住先绘制的物体，即使后绘制的物体在先绘制的物体的“后面”也是如此。使用深度测试可以解决这一问题。使用的方法是：1、以 GL_DEPTH_TEST 为参数调用 glEnable 函数，启动深度测试。2、在必要时（通常是每次绘制画面开始时），清空深度缓冲，即：glClear(GL_DEPTH_BUFFER_BIT);其中，glClear(GL_COLOR_BUFFER_BIT) 与 glClear(GL_DEPTH_BUFFER_BIT)可以合并写为：

glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

且后者的运行速度可能比前者快。

到此为止，我们终于可以得到整个“太阳，地球和月亮”系统的完整代码。

```
// 太阳、地球和月亮

// 假设每个月都是 30 天

// 一年 12 个月，共是 360 天

static int day = 200; // day 的变化：从 0 到 359

void myDisplay(void)

{

   glEnable(GL_DEPTH_TEST);

   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);


   glMatrixMode(GL_PROJECTION);

   glLoadIdentity();

   gluPerspective(75, 1, 1, 400000000);

   glMatrixMode(GL_MODELVIEW);

   glLoadIdentity();

   gluLookAt(0, -200000000, 200000000, 0, 0, 0, 0, 0, 1);
```

```
    // 绘制红色的“太阳”
    glColor3f(1.0f, 0.0f, 0.0f);
    glutSolidSphere(69600000, 20, 20);
    // 绘制蓝色的“地球”
    glColor3f(0.0f, 0.0f, 1.0f);
    glRotatef(day/360.0*360.0, 0.0f, 0.0f, -1.0f);
    glTranslatef(150000000, 0.0f, 0.0f);
    glutSolidSphere(15945000, 20, 20);
    // 绘制黄色的“月亮”
    glColor3f(1.0f, 1.0f, 0.0f);
    glRotatef(day/30.0*360.0 - day/360.0*360.0, 0.0f, 0.0f, -1.0f);
    glTranslatef(38000000, 0.0f, 0.0f);
    glutSolidSphere(4345000, 20, 20);

    glFlush();
}
```

试修改 day 的值，看看画面有何变化。

小结：本课开始，我们正式进入了三维的 OpenGL 世界。

OpenGL 通过矩阵变换来把三维物体转变为二维图象，进而在屏幕上显示出来。为了指定当前操作的是何种矩阵，我们使用了函数 glMatrixMode。

我们可以移动、旋转观察点或者移动、旋转物体，使用的函数是 glTranslate*和 glRotate*。

我们可以缩放物体，使用的函数是 glScale*。

我们可以定义可视空间，这个空间可以是“正投影”的（使用 glOrtho 或 gluOrtho2D），也可以是“透视投影”的（使用 glFrustum 或 gluPerspective）。

我们可以定义绘制到窗口的范围，使用的函数是 glViewport。

矩阵有自己的“堆栈”，方便进行保存和恢复。这在绘制复杂图形时很有帮助。使用的函数是 glPushMatrix 和 glPopMatrix。

好了，艰苦的一课终于完毕。我知道，本课的内容十分枯燥，就连最后的例子也是。但我也没有更好的办法了，希望大家能坚持过去。不必担心，熟悉本课内容后，以后的一段时间内，都会是比较轻松愉快的了。

6. 第六课：

本次课程，我们将进入激动人心的计算机动画世界。

想必大家都知道电影和动画的工作原理吧？是的，快速的把看似连续的画面一幅幅的呈现在人们面前。一旦每秒钟呈现的画面超过 24 幅，人们就会错以为它是连续的。
我们通常观看的电视，每秒播放 25 或 30 幅画面。但对于计算机来说，它可以播放更多的画面，以达到更平滑的效果。如果速度过慢，画面不够平滑。如果速度过快，则人眼未必就能反应得过来。对于一个正常人来说，每秒 60~120 幅图画是比较合适的。具体的数值因人而异。

假设某动画一共有 n 幅画面，则它的工作步骤就是：
显示第 1 幅画面，然后等待一小段时间，直到下一个 1/24 秒
显示第 2 幅画面，然后等待一小段时间，直到下一个 1/24 秒
……
显示第 n 幅画面，然后等待一小段时间，直到下一个 1/24 秒
结束
如果用 C 语言伪代码来描述这一过程，就是：

```
for(i=0; i<n; ++i)
{
  DrawScene(i);
  Wait();
}
```

1、 双缓冲技术

在计算机上的动画与实际的动画有些不同：实际的动画都是先画好了，播放的时候直接拿出来显示就行。计算机动画则是画一张，就拿出来一张，再画下一张，再拿出来。如果所需要绘制的图形很简单，那么这样也没什么问题。但一旦图形比较复杂，绘制需要的时间较长，问题就会变得突出。

让我们把计算机想象成一个画图比较快的人，假如他直接在屏幕上画图，而图形比较复杂，则有可能在他只画了某幅图的一半的时候就被观众看到。而后面虽然他把画补全了，但观众的眼睛却又没有反应过来，还停留在原来那个残缺的画面上。也就是说，有时候观众看到完整的图象，有时却又只看到残缺的图象，这样就造成了屏幕的闪烁。

如何解决这一问题呢？我们设想有两块画板，画图的人在旁边画，画好以后把他手里的画板与挂在屏幕上的画板相交换。这样以来，观众就不会看到残缺的画了。这一技术被应用到计算机图形中，称为双缓冲技术。即：在存储器（很有可能是显存）中开辟两块区域，一块作为发送到显示器的数据，一块作为绘画的区域，在适当的时候交换它们。由于交换两块内存区域实际上只需要交换两个指针，这一方法效率非常高，所以被广泛的采用。

注意：虽然绝大多数平台都支持双缓冲技术，但这一技术并不是 OpenGL 标准中的内容。OpenGL 为了保证更好的可移植性，允许在实现时不使用双缓冲技术。当然，我们常用的 PC 都是支持双缓冲技术的。

要启动双缓冲功能，最简单的办法就是使用 GLUT 工具包。我们以前在 main 函数里面写：

```
glutInitDisplayMode(GLUT_RGB | GLUT_SINGLE);
```

其中 GLUT_SINGLE 表示单缓冲，如果改成 GLUT_DOUBLE 就是双缓冲了。

当然还有需要更改的地方——每次绘制完成时，我们需要交换两个缓冲区，把绘制好的信息用于屏幕显示（否则无论怎么绘制，还是什么都看不到）。如果使用 GLUT 工具包，也可以很轻松的完成这一工作，只要在绘制完成时简单的调用 glutSwapBuffers 函数就可以了。

2、实现连续动画

似乎没有任何疑问，我们应该把绘制动画的代码写成下面这个样子：

```
for(i=0; i<n; ++i)
{
  DrawScene(i);
  glutSwapBuffers();
  Wait();
}
```

但事实上，这样做不太符合窗口系统的程序设计思路。还记得我们的第一个 OpenGL 程序吗？我们在 main 函数里写：glutDisplayFunc(&myDisplay);

意思是对系统说：如果你需要绘制窗口了，请调用 myDisplay 这个函数。为什么我们不直接调用 myDisplay，而要采用这种看似“舍近求远”的做法呢？原因在于——我们自己的程序无法掌握究竟什么时候该绘制窗口。因为一般的窗口系统——拿我们熟悉一点的来说——Windows 和 X 窗口系统，都是支持同时显示多个窗口的。假如你的程序窗口碰巧被别的窗口遮住了，后来用户又把原来遮住的窗口移开，这时你的窗口需要重新绘制。很不幸的，你无法知道这一事件发生的具体时间。因此这一切只好委托操作系统来办了。

现在我们再看上面那个循环。既然 DrawScene 都可以交给操作系统来代办了，那让整个循环运行起来的工作是否也可以交给操作系统呢？答案是肯定的。我们先前的思路是：绘制，然后等待一段时间；再绘制，再等待一段时间。但如果去掉等待的时间，就变成了绘制，绘制，……，不停的绘制。——当然了，资源是公用的嘛，杀毒软件总要工作吧？我的下载不能停下来吧？我的 mp3 播放还不能给耽搁了。总不能因为我们的动画，让其他的工作都停下来。因此，我们需要在 CPU 空闲的时间绘制。

这里的“在 CPU 空闲的时间绘制”和我们在第一课讲的“在需要绘制的时候绘制”有些共通，都是“在 XX 时间做 XX 事”，GLUT 工具包也提供了一个比较类似的函数：glutIdleFunc，表示在 CPU 空闲的时间调用某一函数。其实 GLUT 还提供了一些别的函数，例如“在键盘按下时做某事”等。

到现在，我们已经可以初步开始制作动画了。好的，就拿上次那个“太阳、地球和月亮”的程序开刀，让地球和月亮自己动起来。

```
#include <GL/glut.h>


// 太阳、地球和月亮
// 假设每个月都是 30 天
// 一年 12 个月，共是 360 天
static int day = 200; // day 的变化：从 0 到 359
void myDisplay(void)
{
```

```
    /*******************************************************
     这里的内容照搬上一课的，只因为使用了双缓冲，补上最后这句
    *******************************************************/
    glutSwapBuffers();
}

void myIdle(void)
{
    /* 新的函数，在空闲时调用，作用是把日期往后移动一天并重新绘制，达到动画效果 */
    ++day;
    if( day >= 360 )
        day = 0;
    myDisplay();
}

int main(int argc, char *argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE); // 修改了参数为 GLUT_DOUBLE
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(400, 400);
    glutCreateWindow("太阳，地球和月亮");  // 改了窗口标题
    glutDisplayFunc(&myDisplay);
    glutIdleFunc(&myIdle);              // 新加入了这句
    glutMainLoop();
    return 0;
}
```

3.关于垂直同步

代码是写好了，但相信大家还有疑问。某些朋友可能在运行时发现，虽然 CPU 几乎都用上了，但运动速度很快，根本看不清楚，另一些朋友在运行时发现 CPU 使用率很低，根本就没有把空闲时间完全利用起来。但对于上面那段代码来说，这些现象都是合理的。这里就牵涉到关于垂直同步

的问题。

大家知道显示器的刷新率是比较有限的，一般为 60~120Hz，也就是一秒钟刷新 60~120 次。但如果叫计算机绘制一个简单的画面，例如只有一个三角形，则一秒钟可以绘制成千上万次。因此，如果最大限度的利用计算机的处理能力，绘制很多幅画面，但显示器的刷新速度却跟不上，这不仅造成性能的浪费，还可能带来一些负面影响（例如，显示器只刷新到一半时，需要绘制的内容却变化了，由于显示器是逐行刷新的，于是显示器上半部分和下半部分实际上是来自两幅画面）。采用垂直同步技术可以解决这一问题。即，只有在显示器刷新时，才把绘制好的图象传输出去供显示。这样一来，计算机就不必去绘制大量的根本就用不到的图象了。如果显示器的刷新率为 85Hz，则计算机一秒钟只需要绘制 85 幅图象就足够，如果场景足够简单，就会造成比较多的 CPU 空闲。

几乎所有的显卡都支持“垂直同步”这一功能。

垂直同步也有它的问题。如果刷新频率为 60Hz，则在绘制比较简单的场景时，绘制一幅图画需要的时间很段，帧速可以恒定在 60FPS（即 60 帧/秒）。如果场景变得复杂，绘制一幅图画的时间超过了 1/60 秒，则帧速将急剧下降。

如果绘制一幅图画的时间为 1/50，则在第一个 1/60 秒时，显示器需要刷新了，但由于新的图画没有画好，所以只能显示原来的图画，等到下一个 1/60 秒时才显示新的图画。于是显示一幅图画实际上用了 1/30 秒，帧速为 30FPS。（如果不采用垂直同步，则帧速应该是 50FPS）

如果绘制一幅图画的时间更长，则下降的趋势就是阶梯状的：60FPS，30FPS，20FPS，……（60/1，60/2，60/3，……）

如果每一幅图画的复杂程度是不一致的，且绘制它们需要的时间都在 1/60 上下。则在 1/60 时间内画完时，帧速为 60FPS，在 1/60 时间未完成时，帧速为 30FPS，这就造成了帧速的跳动。这是很麻烦的事情，需要避免它——要么想办法简化每一画面的绘制时间，要么都延迟一小段时间，以作到统一。

回过头来看前面的问题。如果使用了大量的 CPU 而且速度很快无法看清，则打开垂直同步可以解决该问题。当然如果你认为垂直同步有这样那样的缺点，也可以关闭它。——至于如何打开和关闭，因操作系统而异了。具体步骤请自己搜索之。

当然，也有其它办法可以控制动画的帧速，或者尽量让动画的速度尽量和帧速无关。不过这里面很多内容都是与操作系统比较紧密的，况且它们跟 OpenGL 关系也不太大。这里就不做介绍了。

4、计算帧速

不知道大家玩过 3D Mark 这个软件没有，它可以运行各种场景，测出帧速，并且为你的系统给出评分。这里我也介绍一个计算帧速的方法。

根据定义，帧速就是一秒钟内播放的画面数目（FPS）。我们可以先测量绘制两幅画面之间时间 t，然后求它的倒数即可。假如 t=0.05s，则 FPS 的值就是 1/0.05=20。

理论上是如此了，可是如何得到这个时间呢？通常 C 语言的 time 函数精确度一般只到一秒，肯定是不行了。clock 函数也就到十毫秒左右，还是有点不够。因为 FPS 为 60 和 FPS 为 100 的时候，t 的值都是十几毫秒。你知道如何测量一张纸的厚度吗？一个粗略的办法就是：用很多张纸叠在一起测厚度，计算平均值就可以了。我们这里也可以这样办。测量绘制 50 幅画面（包括垂直同步等因素的等待时间）需要的时间 t'，由 t'

=t*50 很容易的得到 FPS=1/t=50/t'

下面这段代码可以统计该函数自身的调用频率，（原理就像上面说的那样），程序并不复杂，并且这并不属于 OpenGL 的内容，所以我不打算详细讲述它。

```
#include <time.h>
double CalFrequency()
{
   static int count;
   static double save;
   static clock_t last, current;
   double timegap;

   ++count;
   if( count <= 50 )
      return save;
   count = 0;
   last = current;
   current = clock();
   timegap = (current-last)/(double)CLK_TCK;
   save = 50.0/timegap;
   return save;
}
```

最后，要把计算的帧速显示出来，但我们并没有学习如何使用 OpenGL 把文字显示到屏幕上。——但不要忘了，在我们的图形窗口背后，还有一个命令行窗口~使用 printf 函数就可以轻易的输出文字了。

```
#include <stdio.h>

double FPS = CalFrequency();
printf("FPS = %f\n", FPS);
```

最后的一步，也被我们解决了——虽然做法不太雅观，没关系，以后我们还会改善它的。

时间过得太久，每次给的程序都只是一小段，一些朋友难免会出问题。

现在，我给出一个比较完整的程序，供大家参考。

```
#include <GL/glut.h>
#include <stdio.h>
#include <time.h>

// 太阳、地球和月亮
// 假设每个月都是 12 天
// 一年 12 个月，共是 360 天
static int day = 200; // day 的变化：从 0 到 359

double CalFrequency()
{
   static int count;
   static double save;
   static clock_t last, current;
   double timegap;

   ++count;
   if( count <= 50 )
      return save;
   count = 0;
   last = current;
   current = clock();
   timegap = (current-last)/(double)CLK_TCK;
   save = 50.0/timegap;
   return save;
}

void myDisplay(void)
{
   double FPS = CalFrequency();
```

```
    printf("FPS = %f\n", FPS);

    glEnable(GL_DEPTH_TEST);
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(75, 1, 1, 400000000);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0, -200000000, 200000000, 0, 0, 0, 0, 0, 1);

    // 绘制红色的“太阳”
    glColor3f(1.0f, 0.0f, 0.0f);
    glutSolidSphere(69600000, 20, 20);
    // 绘制蓝色的“地球”
    glColor3f(0.0f, 0.0f, 1.0f);
    glRotatef(day/360.0*360.0, 0.0f, 0.0f, -1.0f);
    glTranslatef(150000000, 0.0f, 0.0f);
    glutSolidSphere(15945000, 20, 20);
    // 绘制黄色的“月亮”
    glColor3f(1.0f, 1.0f, 0.0f);
    glRotatef(day/30.0*360.0 - day/360.0*360.0, 0.0f, 0.0f, -1.0f);
    glTranslatef(38000000, 0.0f, 0.0f);
    glutSolidSphere(4345000, 20, 20);

    glFlush();
    glutSwapBuffers();
}
```

```
void myIdle(void)
{
   ++day;
   if( day >= 360 )
      day = 0;
   myDisplay();
}

int main(int argc, char *argv[])
{
   glutInit(&argc, argv);
   glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);
   glutInitWindowPosition(100, 100);
   glutInitWindowSize(400, 400);
   glutCreateWindow("太阳，地球和月亮");
   glutDisplayFunc(&myDisplay);
   glutIdleFunc(&myIdle);
   glutMainLoop();
   return 0;
}
```

小结：

OpenGL 动画和传统意义上的动画相似，都是把画面一幅一幅的呈现在观众面前。一旦画面变换的速度快了，观众就会认为画面是连续的。

双缓冲技术是一种在计算机图形中普遍采用的技术，绝大多数 OpenGL 实现都支持双缓冲技术。

通常都是利用 CPU 空闲的时候绘制动画，但也可以有其它的选择。

介绍了垂直同步的相关知识。

介绍了一种简单的计算帧速（FPS）的方法。

最后，我们列出了一份完整的天体动画程序清单。

7. 第七课：

从生理学的角度上讲，眼睛之所以看见各种物体，是因为光线直接或间接的从它们那里到达了眼睛。人类对于光线强弱的变化的反应，比对于颜色变化的反应来得灵敏。因此对于人类而言，光线很大程度上表现了物体的立体感。

请看图 1，图中绘制了两个大小相同的白色球体。其中右边的一个是没有使用任何光照效果的，

它看起来就像是一个二维的圆盘，没有立体的感觉。左边的一个是使用了简单的光照效果的，我们通过光照的层次，很容易的认为它是一个三维的物体。

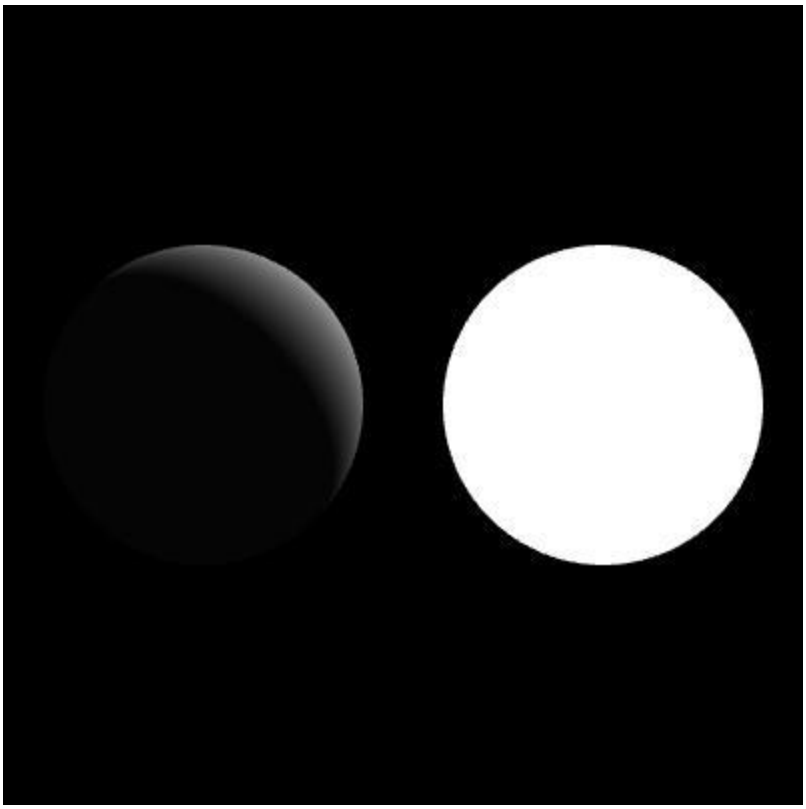


图 1

OpenGL 对于光照效果提供了直接的支持，只需要调用某些函数，便可以实现简单的光照效果。但是在这之前，我们有必要了解一些基础知识。

一、 建立光照模型

在现实生活中，某些物体本身就会发光，例如太阳、电灯等，而其它物体虽然不会发光，但可以反射来自其它物体的光。这些光通过各种方式传播，最后进入我们的眼睛——于是一幅画面就在我们的眼中形成了。

就目前的计算机而言，要准确模拟各种光线的传播，这是无法做到的事情。比如一个四面都是粗糙墙壁的房间，一盏电灯所发出的光线在很短的时间内就会经过非常多次的反射，最终几乎布满了房间的每一个角落，这一过程即使使用目前运算速度最快的计算机，也无法精确模拟。不过，我们并不需要精确的模拟各种光线，只需要找到一种近似的计算方式，使它的最终结果让我们的眼睛认为它是真实的，这就可以了。

OpenGL 在处理光照时采用这样一种近似：把光照系统分为三部分，分别是光源、材质和光照环境。光源就是光的来源，可以是前面所说的太阳或者电灯等。材质是指接受光照的各种物体的表面，由于物体如何反射光线只由物体表面决定（OpenGL 中没有考虑光的折射），材质特点就决定了物体反射光线的特点。光照环境是指一些额外的参数，它们将影响最终的光照画面，比如一些光线经过多次反射后，已经无法分清它究竟是由哪个光源发出，这时，指定一个“环境亮度”参数，可以使最后形成的画面更接近于真实情况。

在物理学中，光线如果射入理想的光滑平面，则反射后的光线是很规则的（这样的反射称为镜面反射）。光线如果射入粗糙的、不光滑的平面，则反射后的光线是杂乱的（这样

的反射称为漫反射）。现实生活中的物体在反射光线时，并不是绝对的镜面反射或漫反射，但可以看成是这两种反射的叠加。对于光源发出的光线，可以分别设置其经过镜面反射和漫反射后的光线强度。对于被光线照射的材质，也可以分别设置光线经过镜面反射和漫反射后的光线强度。这些因素综合起来，就形成了最终的光照效果。

二、法线向量

根据光的反射定律，由光的入射方向和入射点的法线就可以得到光的出射方向。因此，对于指定的物体，在指定了光源后，即可计算出光的反射方向，进而计算出光照效果的画面。在 OpenGL 中，法线的方向是用一个向量来表示。

不幸的是，OpenGL 并不会根据你所指定的多边形各个顶点来计算出这些多边形所构成的物体的表面的每个点的法线（这话听着有些迷糊），通常，为了实现光照效果，需要在代码中为每一个顶点指定其法线向量。

指定法线向量的方式与指定颜色的方式有雷同之处。在指定颜色时，只需要指定每一个顶点的颜色，OpenGL 就可以自行计算顶点之间的其它点的颜色。并且，颜色一旦被指定，除非再指定新的颜色，否则以后指定的所有顶点都将以这一向量作为自己的颜色。

在指定法线向量时，只需要指定每一个顶点的法线向量，OpenGL 会自行计算顶点之间的其它点的法线向量。并且，法线向量一旦被指定，除非再指定新的法线向量，否则以后指定的所有顶点都将以这一向量作为自己的法线向量。使用 glColor*函数可以指定颜色，而使用 glNormal*函数则可以指定法线向量。

注意：使用 glTranslate*函数或者 glRotate*函数可以改变物体的外观，但法线向量并不会随之改变。然而，使用 glScale*函数，对每一坐标轴进行不同程度的缩放，很有可能导致法线向量的不正确，虽然 OpenGL 提供了一些措施来修正这一问题，但由此也带来了各种开销。因此，在使用了法线向量的场合，应尽量避免使用 glScale*函数。即使使用，也最好保证各坐标轴进行等比例缩放。

三. 控制光源

在 OpenGL 中，仅仅支持有限数量的光源。使用 GL_LIGHT0 表示第 0 号光源，GL_LIGHT1 表示第 1 号光源，依次类推，OpenGL 至少会支持 8 个光源，即 GL_LIGHT0 到 GL_LIGHT7。使用 glEnable 函数可以开启它们。例如，glEnable(GL_LIGHT0);可以开启第 0 号光源。使用 glDisable 函数则可以关闭光源。一些 OpenGL 实现可能支持更多数量的光源，但总的来说，开启过多的光源将会导致程序运行速度的严重下降，玩过 3D Mark 的朋友可能多少也有些体会。一些场景中可能有成百上千的电灯，这时可能需要采取一些近似的手段来进行编程，否则以目前的计算机而言，是无法运行这样的程序的。

每一个光源都可以设置其属性，这一动作是通过 glLight*函数完成的。glLight*函数具有三个参数，第一个参数指明是设置哪一个光源的属性，第二个参数指明是设置该光源的哪一个属性，第三个参数则是指明把该属性值设置成多少。光源的属性众多，下面将分别介绍。

（1）GL_AMBIENT、GL_DIFFUSE、GL_SPECULAR 属性。这三个属性表示了光源所发出的光的反射特性（以及颜色）。每个属性由四个值表示，分别代表了颜色的 R, G, B, A 值。GL_AMBIENT 表示该光源所发出的光，经过非常多次的反射后，最终遗留在整个光照环境中的强度（颜色）。GL_DIFFUSE 表示该光源所发出的光，照射到粗糙表面时经过漫反射，所得到的光的强度（颜色）。GL_SPECULAR 表示该光源所发出的光，照射到光滑表面时经过镜面反射，所得到的光的强度（颜色）。

（2）GL_POSITION 属性。表示光源所在的位置。由四个值（X, Y, Z, W）表示。如果第四个值 W 为零，则表示该光源位于无限远处，前三个值表示了它所在的方向。这种光源称为方向性光源，通常，太阳可以近似的被认为是方向性光源。如果第四个值 W 不为零，则 X/W, Y/W, Z/W 表示了

光源的位置。这种光源称为位置性光源。对于位置性光源，设置其位置与设置多边形顶点的方式相似，各种矩阵变换函数例如：glTranslate*、glRotate*等在这里也同样有效。方向性光源在计算时比位置性光源快了不少，因此，在视觉效果允许的情况下，应该尽可能的使用方向性光源。

（3）GL_SPOT_DIRECTION、GL_SPOT_EXPONENT、GL_SPOT_CUTOFF 属性。表示将光源作为聚光灯使用（这些属性只对位置性光源有效）。很多光源都是向四面八方发射光线，但有时候一些光源则是只向某个方向发射，比如手电筒，只向一个较小的角度发射光线。GL_SPOT_DIRECTION 属性有三个值，表示一个向量，即光源发射的方向。GL_SPOT_EXPONENT 属性只有一个值，表示聚光的程度，为零时表示光照范围内向各方向发射的光线强度相同，为正数时表示光照向中央集中，正对发射方向的位置受到更多光照，其它位置受到较少光照。数值越大，聚光效果就越明显。GL_SPOT_CUTOFF 属性也只有一个值，表示一个角度，它是光源发射光线所覆盖角度的一半（见图 2），其取值范围在 0 到 90 之间，也可以取 180 这个特殊值。取值为 180 时表示光源发射光线覆盖 360 度，即不使用聚光灯，向全周围发射。

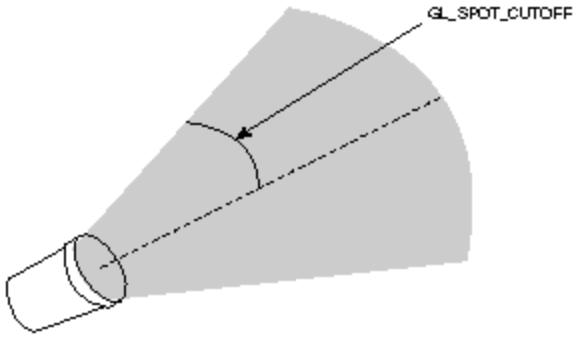


图 2

（4）GL_CONSTANT_ATTENUATION、GL_LINEAR_ATTENUATION、GL_QUADRATIC_ATTENUATION 属性。这三个属性表示了光源所发出的光线的直线传播特性（这些属性只对位置性光源有效）。现实生活中，光线的强度随着距离的增加而减弱，OpenGL 把这个减弱的趋势抽象成函数：

衰减因子 = 1 / (k1 + k2 * d + k3 * k3 * d)

其中 d 表示距离，光线的初始强度乘以衰减因子，就得到对应距离的光线强度。k1, k2, k3 分别就是

GL_CONSTANT_ATTENUATION, GL_LINEAR_ATTENUATION, GL_QUADRATIC_ATTENUATION。通过设置这三个常数，就可以控制光线在传播过程中的减弱趋势。

属性还真是不少。当然了，如果是使用方向性光源，（3）（4）这两类属性就不会用到了，问题就变得简单明了。

四. 控制材质

材质与光源相似，也需要设置众多的属性。不同的是，光源是通过 glLight*函数来设置的，而材质则是通过 glMaterial*函数来设置的。

glMaterial*函数有三个参数。第一个参数表示指定哪一面的属性。可以是 GL_FRONT、GL_BACK 或者 GL_FRONT_AND_BACK。分别表示设置“正面”“背面”的材质，或者两面同时设置。（关于“正面”“背面”的内容需要参看前些课程的内容）第二、第三个参数与 glLight*函数的第二、三个参数作用类似。下面分别说明 glMaterial*函数可以指定的材质属性。

（1）GL_AMBIENT、GL_DIFFUSE、GL_SPECULAR 属性。这三个属性与光源的三个

对应属性类似，每一属性都由四个值组成。GL_AMBIENT 表示各种光线照射到该材质上，经过很多次反射后最终遗留在环境中的光线强度（颜色）。GL_DIFFUSE 表示光线照射到该材质上，经过漫反射后形成的光线强度（颜色）。GL_SPECULAR 表示光线照射到该材质上，经过镜面反射后形成的光线强度（颜色）。通常，GL_AMBIENT 和 GL_DIFFUSE 都取相同的值，可以达到比较真实的效果。使用 GL_AMBIENT_AND_DIFFUSE 可以同时设置 GL_AMBIENT 和 GL_DIFFUSE 属性。

（2）GL_SHININESS 属性。该属性只有一个值，称为“镜面指数”，取值范围是 0 到 128。该值越小，表示材质越粗糙，点光源发射的光线照射到上面，也可以产生较大的亮点。该值越大，表示材质越类似于镜面，光源照射到上面后，产生较小的亮点。

（3）GL_EMISSION 属性。该属性由四个值组成，表示一种颜色。OpenGL 认为该材质本身就微微的向外发射光线，以至于眼睛感觉到它有这样的颜色，但这光线又比较微弱，以至于不会影响到其它物体的颜色。

（4）GL_COLOR_INDEXES 属性。该属性仅在颜色索引模式下使用，由于颜色索引模式下的光照比 RGBA 模式要复杂，并且使用范围较小，这里不做讨论。

五、选择光照模型

这里所说的“光照模型”是 OpenGL 的术语，它相当于我们在前面提到的“光照环境”。在 OpenGL 中，光照模型包括四个部分的内容：全局环境光线（即那些充分散射，无法分清究竟来自哪个光源的光线）的强度、观察点位置是在较近位置还是在无限远处、物体正面与背面是否分别计算光照、镜面颜色（即 GL_SPECULAR 属性所指定的颜色)的计算是否从其它光照计算中分离出来，并在纹理操作以后在进行应用。

以上四方面的内容都通过同一个函数 glLightModel*来进行设置。该函数有两个参数，第一个表示要设置的项目，第二个参数表示要设置成的值。

GL_LIGHT_MODEL_AMBIENT 表示全局环境光线强度，由四个值组成。

GL_LIGHT_MODEL_LOCAL_VIEWER 表示是否在近处观看，若是则设置为 GL_TRUE，否则（即在无限远处观看）设置为 GL_FALSE。

GL_LIGHT_MODEL_TWO_SIDE 表示是否执行双面光照计算。如果设置为 GL_TRUE，则 OpenGL 不仅将根据法线向量计算正面的光照，也会将法线向量反转并计算背面的光照。

GL_LIGHT_MODEL_COLOR_CONTROL 表示颜色计算方式。如果设置为 GL_SINGLE_COLOR，表示按通常顺序操作，先计算光照，再计算纹理。如果设置为 GL_SEPARATE_SPECULAR_COLOR，表示将 GL_SPECULAR 属性分离出来，先计算光照的其它部分，待纹理操作完成后再计算 GL_SPECULAR。后者通常可以使画面效果更为逼真（当然，如果本身就没有执行任何纹理操作，这样的分离就没有任何意义）。

六、最后的准备

到现在可以说是完事俱备了。不过，OpenGL 默认是关闭光照处理的。要打开光照处理功能，使用下面的语句：

```
glEnable(GL_LIGHTING);
```

要关闭光照处理功能，使用 glDisable(GL_LIGHTING);即可。

七、示例程序

到现在，我们已经可以编写简单的使用光照的 OpenGL 程序了。

我们仍然以太阳、地球作为例子（这次就不考虑月亮了^-^），把太阳作为光源，模拟地球围绕太阳转动时光照的变化。于是，需要设置一个光源——太阳，设置两种材质——太阳的材质和地球的材质。把太阳光线设置为白色，位置在画面正中。把太阳的材质设置为微微散发出红色的光芒，把地球的材质设置为微微散发出暗淡的蓝色光芒，并且反射蓝色的光芒，镜面指数设置成一个比

较小的值。简单起见，不再考虑太阳和地球的大小关系，用同样大小的球体来代替之。
关于法线向量。球体表面任何一点的法线向量，就是球心到该点的向量。如果使用 glutSolidSphere 函数来绘制球体，则该函数会自动的指定这些法线向量，不必再手工指出。如果是自己指定若干的顶点来绘制一个球体，则需要自己指定法线响亮。
由于我们使用的太阳是一个位置性光源，在设置它的位置时，需要利用到矩阵变换。因此，在设置光源的位置以前，需要先设置好各种矩阵。利用 gluPerspective 函数来创建具有透视效果的视图。我们也将利用前面课程所学习的动画知识，让整个画面动起来。

下面给出具体的代码：

```
#include <gl/glut.h>

#define WIDTH 400
#define HEIGHT 400

static GLfloat angle = 0.0f;

void myDisplay(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

  // 创建透视效果视图
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  gluPerspective(90.0f, 1.0f, 1.0f, 20.0f);
  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  gluLookAt(0.0, 5.0, -10.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

  // 定义太阳光源，它是一种白色的光源
  {
  GLfloat sun_light_position[] = {0.0f, 0.0f, 0.0f, 1.0f};
  GLfloat sun_light_ambient[]  = {0.0f, 0.0f, 0.0f, 1.0f};
  GLfloat sun_light_diffuse[]  = {1.0f, 1.0f, 1.0f, 1.0f};
  GLfloat sun_light_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};

  glLightfv(GL_LIGHT0, GL_POSITION, sun_light_position);
  glLightfv(GL_LIGHT0, GL_AMBIENT,  sun_light_ambient);
  glLightfv(GL_LIGHT0, GL_DIFFUSE,  sun_light_diffuse);
  glLightfv(GL_LIGHT0, GL_SPECULAR, sun_light_specular);

  glEnable(GL_LIGHT0);
  glEnable(GL_LIGHTING);
  glEnable(GL_DEPTH_TEST);
  }
```

```
    // 定义太阳的材质并绘制太阳
    {
        GLfloat sun_mat_ambient[]  = {0.0f, 0.0f, 0.0f, 1.0f};
        GLfloat sun_mat_diffuse[]  = {0.0f, 0.0f, 0.0f, 1.0f};
        GLfloat sun_mat_specular[] = {0.0f, 0.0f, 0.0f, 1.0f};
        GLfloat sun_mat_emission[] = {0.5f, 0.0f, 0.0f, 1.0f};
        GLfloat sun_mat_shininess  = 0.0f;

        glMaterialfv(GL_FRONT, GL_AMBIENT,   sun_mat_ambient);
        glMaterialfv(GL_FRONT, GL_DIFFUSE,   sun_mat_diffuse);
        glMaterialfv(GL_FRONT, GL_SPECULAR,  sun_mat_specular);
        glMaterialfv(GL_FRONT, GL_EMISSION,  sun_mat_emission);
        glMaterialf (GL_FRONT, GL_SHININESS, sun_mat_shininess);

        glutSolidSphere(2.0, 40, 32);
    }

    // 定义地球的材质并绘制地球
    {
        GLfloat earth_mat_ambient[]  = {0.0f, 0.0f, 0.5f, 1.0f};
        GLfloat earth_mat_diffuse[]  = {0.0f, 0.0f, 0.5f, 1.0f};
        GLfloat earth_mat_specular[] = {0.0f, 0.0f, 1.0f, 1.0f};
        GLfloat earth_mat_emission[] = {0.0f, 0.0f, 0.0f, 1.0f};
        GLfloat earth_mat_shininess  = 30.0f;

        glMaterialfv(GL_FRONT, GL_AMBIENT,   earth_mat_ambient);
        glMaterialfv(GL_FRONT, GL_DIFFUSE,   earth_mat_diffuse);
        glMaterialfv(GL_FRONT, GL_SPECULAR,  earth_mat_specular);
        glMaterialfv(GL_FRONT, GL_EMISSION,  earth_mat_emission);
        glMaterialf (GL_FRONT, GL_SHININESS, earth_mat_shininess);

        glRotatef(angle, 0.0f, -1.0f, 0.0f);
        glTranslatef(5.0f, 0.0f, 0.0f);
        glutSolidSphere(2.0, 40, 32);
    }

    glutSwapBuffers();
}
void myIdle(void)
{
    angle += 1.0f;
    if( angle >= 360.0f )
        angle = 0.0f;
```

```
    myDisplay();
}

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutInitWindowPosition(200, 200);
    glutInitWindowSize(WIDTH, HEIGHT);
    glutCreateWindow("OpenGL 光照演示");
    glutDisplayFunc(&myDisplay);
    glutIdleFunc(&myIdle);
    glutMainLoop();
    return 0;
}
```

小结：

本课介绍了 OpenGL 光照的基本知识。OpenGL 把光照分解为光源、材质、光照模式三个部分，根据这三个部分的各种信息，以及物体表面的法线向量，可以计算得到最终的光照效果。

光源、材质和光照模式都有各自的属性，尽管属性种类繁多，但这些属性都只用很少的几个函数来设置。使用 glLight*函数可设置光源的属性，使用 glMaterial*函数可设置材质的属性，使用 glLightModel*函数可设置光照模式。

GL_AMBIENT、GL_DIFFUSE、GL_SPECULAR 这三种属性是光源和材质所共有的，如果某光源发出的光线照射到某材质的表面，则最终的漫反射强度由两个 GL_DIFFUSE 属性共同决定，最终的镜面反射强度由两个 GL_SPECULAR 属性共同决定。

可以使用多个光源来实现各种逼真的效果，然而，光源数量的增加将造成程序运行速度的明显下降。

在使用 OpenGL 光照过程中，属性的种类和数量都非常繁多，通常，需要很多的经验才可以熟练的设置各种属性，从而形成逼真的光照效果。（各位也看到了，其实这个课程的示例程序中，属性设置也不怎么好）。然而，设置这些属性的艺术性远远超过了技术性，往往是一些美术制作人员设置好各种属性（并保存为文件），然后由程序员编写的程序去执行绘制工作。因此，即使目前无法熟练运用各种属性，也不必过于担心。如果条件允许，可以玩玩类似 3DS MAX 之类的软件，对理解光照、熟悉各种属性设置会有一些帮助。

在课程的最后，我们给出了一个样例程序，演示了太阳和地球模型中的光照效果。

8. 第八课：

今天介绍关于 OpenGL 显示列表的知识。本课内容并不多，但需要一些理解能力。在学习时，可以将显示列表与 C 语言的“函数”进行类比，加深体会。

我们已经知道，使用 OpenGL 其实只要调用一系列的 OpenGL 函数就可以了。然而，这种方式在一些时候可能导致问题。比如某个画面中，使用了数千个多边形来表现一个比较真实的人物，OpenGL 为了产生这数千个多边形，就需要不停的调用 glVertex*函数，每一个多边形将至少调用三次（因为多边形至少有三个顶点），于是绘制一个比较真实的人物就需要调用上万次的 glVertex*函数。更糟糕的是，如果我们需要每秒钟绘制 60 幅画面，则每秒调用的 glVertex*函数次数就会超过数十万次，乃至接近百万次。这样的情况

是我们所不愿意看到的。

同时，考虑这样一段代码：

```
const int segments = 100;
const GLfloat pi = 3.14f;
int i;
glLineWidth(10.0);
glBegin(GL_LINE_LOOP);
for(i=0; i<segments; ++i)
{
   GLfloat tmp = 2 * pi * i / segments;
   glVertex2f(cos(tmp), sin(tmp));
}
glEnd();
```

这段代码将绘制一个圆环。如果我们在每次绘制图象时调用这段代码，则虽然可以达到绘制圆环的目的，但是 cos、sin 等开销较大的函数被多次调用，浪费了 CPU 资源。如果每一个顶点不是通过 cos、sin 等函数得到，而是使用更复杂的运算方式来得到，则浪费的现象就更加明显。

经过分析，我们可以发现上述两个问题的共同点：程序多次执行了重复的工作，导致 CPU 资源浪费和运行速度的下降。使用显示列表可以较好的解决上述两个问题。

在编写程序时，遇到重复的工作，我们往往是将重复的工作编写为函数，在需要的地方调用它。类似的，在编写 OpenGL 程序时，遇到重复的工作，可以创建一个显示列表，把重复的工作装入其中，并在需要的地方调用这个显示列表。

使用显示列表一般有四个步骤：分配显示列表编号、创建显示列表、调用显示列表、销毁显示列表。

一、分配显示列表编号

OpenGL 允许多个显示列表同时存在，就好象 C 语言允许程序中有多个函数同时存在。C 语言中，不同的函数用不同的名字来区分，而在 OpenGL 中，不同的显示列表用不同的正整数来区分。

你可以自己指定一些各不相同的正整数来表示不同的显示列表。但是如果你不够小心，可能出现一个显示列表将另一个显示列表覆盖的情况。为了避免这一问题，使用 glGenLists 函数来自动分配一个没有使用的显示列表编号。

glGenLists 函数有一个参数 i，表示要分配 i 个连续的未使用的显示列表编号。返回的是分配的若干连续编号中最小的一个。例如，glGenLists(3);如果返回 20，则表示分配了 20、21、22 这三个连续的编号。如果函数返回零，表示分配失败。

可以使用 glIsList 函数判断一个编号是否已经被用作显示列表。

二、创建显示列表

创建显示列表实际上就是把各种 OpenGL 函数的调用装入到显示列表中。使用 glNewList 开始装入，使用 glEndList 结束装入。glNewList 有两个参数，第一个参数是一个正整数表示装入到哪个显示列表。第二个参数有两种取值，如果为 GL_COMPILE，则表示以下的内容只是装入到显示列表，但现在不执行它们；如果为 GL_COMPILE_AND_EXECUTE，表示在装入的同时，把装入的内容执行一遍。

例如，需要把“设置颜色为红色，并且指定一个坐标为(0, 0)的顶点”这两条命令装入到编号为 list 的显示列表中，并且在装入的时候不执行，则可以用下面的代码：

```
glNewList(list, GL_COMPILE);
glColor3f(1.0f, 0.0f, 0.0f);
glVertex2f(0.0f, 0.0f);
glEnd();
```

注意：显示列表只能装入 OpenGL 函数，而不能装入其它内容。例如：

```
int i = 3;
glNewList(list, GL_COMPILE);
if( i > 20 )
   glColor3f(1.0f, 0.0f, 0.0f);
glVertex2f(0.0f, 0.0f);
glEnd();
```

其中 if 这个判断就没有被装入到显示列表。以后即使修改 i 的值，使 i>20 的条件成立，则 glColor3f 这个函数也不会被执行。因为它根本就不存在于显示列表中。

另外，并非所有的 OpenGL 函数都可以装入到显示列表中。例如，各种用于查询的函数，它们无法被装入到显示列表，因为它们都具有返回值，而 glCallList 和 glCallLists 函数都不知道如何处理这些返回值。在网络方式下，设置客户端状态的函数也无法被装入到显示列表，这是因为显示列表被保存到服务器端，各种设置客户端状态的函数在发送到服务器端以前就被执行了，而服务器端无法执行这些函数。分配、创建、删除显示列表的动作也无法被装入到另一个显示列表，但调用显示列表的动作则可以被装入到另一个显示列表。

三、调用显示列表

使用 glCallList 函数可以调用一个显示列表。该函数有一个参数，表示要调用的显示列表的编号。例如，要调用编号为 10 的显示列表，直接使用 glCallList(10);就可以了。

使用 glCallLists 函数可以调用一系列的显示列表。该函数有三个参数，第一个参数表示了要调用多少个显示列表。第二个参数表示了这些显示列表的编号的储存格式，可以是 GL_BYTE（每个编号用一个 GLbyte 表示），GL_UNSIGNED_BYTE（每个编号用一个 GLubyte 表示），GL_SHORT，GL_UNSIGNED_SHORT，GL_INT，GL_UNSIGNED_INT，GL_FLOAT。第三个参数表示了这些显示列表的编号所在的位置。在使用该函数前，需要用 glListBase 函数来设置一个偏移量。假设偏移量为 k，且 glCallLists 中要求调用的显示列表编号依次为 l1, l2, l3, ...，则实际调用的显示列表为 l1+k, l2+k, l3+k, ...。

例如：

```
GLuint lists[] = {1, 3, 4, 8};
glListBase(10);
glCallLists(4, GL_UNSIGNED_INT, lists);
```

则实际上调用的是编号为 11, 13, 14, 18 的四个显示列表。

注：“调用显示列表”这个动作本身也可以被装在另一个显示列表中。

四、销毁显示列表

销毁显示列表可以回收资源。使用 glDeleteLists 来销毁一串编号连续的显示列表。

例如，使用 glDeleteLists(20, 4);将销毁 20，21，22，23 这四个显示列表。

使用显示列表将会带来一些开销，例如，把各种动作保存到显示列表中会占用一定数量的内存资源。但如果使用得当，显示列表可以提升程序的性能。这主要表现在以下方面：

1、明显的减少 OpenGL 函数的调用次数。如果函数调用是通过网络进行的（Linux 等操作系统支持这样的方式，即由应用程序在客户端发出 OpenGL 请求，由网络上的另一台服务器进行实际的绘图操作），将显示列表保存在服务器端，可以大大减少网络负担。

2、保存中间结果，避免一些不必要的计算。例如前面的样例程序中，cos、sin 函数的计算结果被直接保存到显示列表中，以后使用时就不必重复计算。

3、便于优化。我们已经知道，使用 glTranslate*、glRotate*、glScale*等函数时，实际上是执行矩阵乘法操作，由于这些函数经常被组合在一起使用，通常会出现矩阵的连乘。这时，如果把这些操作保存到显示列表中，则一些复杂的 OpenGL 版本会尝试先计算出连乘的一部分结果，从而提高程序的运行速度。在其它方面也可能存在类似的例子。

同时，显示列表也为程序的设计带来方便。我们在设置一些属性时，经常把一些相关的函数放在一起调用，（比如，把设置光源的各种属性的函数放到一起）这时，如果把这些设置属性的操作装入到显示列表中，则可以实现属性的成组的切换。

当然了，即使使用显示列表在某些情况下可以提高性能，但这种提高很可能并不明显。毕竟，在硬件配置和大致的软件算法都不变的前提下，性能可提升的空间并不大。

显示列表的内容就是这么多了，下面我们看一个例子。

假设我们需要绘制一个旋转的彩色正四面体，则可以这样考虑：设置一个全局变量 angle，然后让它的值不断的增加（到达 360 后又恢复为 0，周而复始）。每次需要绘制图形时，根据 angle 的值进行旋转，然后绘制正四面体。这里正四面体采用显示列表来实现，即把绘制正四面体的若干 OpenGL 函数装到一个显示列表中，然后每次需要绘制时，调用这个显示列表即可。

将正四面体的四个顶点颜色分别设置为红、黄、绿、蓝，通过数学计算，将坐标设置为：

(-0.5, -5*sqrt(5)/48, sqrt(3)/6),

(0.5, -5*sqrt(5)/48, sqrt(3)/6),

(0, -5*sqrt(5)/48, -sqrt(3)/3),

(0, 11*sqrt(6)/48, 0)

2007 年 4 月 24 日修正：以上结果有误，通过计算 AB, AC, AD, BC, BD, CD 的长度，发现 AD, BD, CD 的长度与 1.0 有较大偏差。正确的坐标应该是：

A 点：(0.5, -sqrt(6)/12, -sqrt(3)/6)

B 点：(-0.5, -sqrt(6)/12, -sqrt(3)/6)

C 点：(0, -sqrt(6)/12, sqrt(3)/3)

D 点：(0, sqrt(6)/4, 0)

程序代码中也做了相应的修改

下面给出程序代码，大家可以从中体会一下显示列表的用法。

```
#include <gl/glut.h>

#define WIDTH 400
#define HEIGHT 400

#include <math.h>
#define ColoredVertex(c, v) do{ glColor3fv(c); glVertex3fv(v); }while(0)

GLfloat angle = 0.0f;

void myDisplay(void)
{
   static int list = 0;
   if( list == 0 )
   {
      // 如果显示列表不存在，则创建
      /* GLfloat
         PointA[] = {-0.5, -5*sqrt(5)/48,  sqrt(3)/6},
         PointB[] = { 0.5, -5*sqrt(5)/48,  sqrt(3)/6},
         PointC[] = {   0, -5*sqrt(5)/48, -sqrt(3)/3},
         PointD[] = {   0, 11*sqrt(6)/48,         0}; */
      // 2007 年 4 月 27 日修改
      GLfloat
         PointA[] = { 0.5f, -sqrt(6.0f)/12, -sqrt(3.0f)/6},
         PointB[] = {-0.5f, -sqrt(6.0f)/12, -sqrt(3.0f)/6},
         PointC[] = { 0.0f, -sqrt(6.0f)/12,  sqrt(3.0f)/3},
```

```
    PointD[] = { 0.0f,   sqrt(6.0f)/4,          0};
GLfloat
    ColorR[] = {1, 0, 0},
    ColorG[] = {0, 1, 0},
    ColorB[] = {0, 0, 1},
    ColorY[] = {1, 1, 0};

list = glGenLists(1);
glNewList(list, GL_COMPILE);
glBegin(GL_TRIANGLES);
// 平面 ABC
ColoredVertex(ColorR, PointA);
ColoredVertex(ColorG, PointB);
ColoredVertex(ColorB, PointC);
// 平面 ACD
ColoredVertex(ColorR, PointA);
ColoredVertex(ColorB, PointC);
ColoredVertex(ColorY, PointD);
// 平面 CBD
ColoredVertex(ColorB, PointC);
ColoredVertex(ColorG, PointB);
ColoredVertex(ColorY, PointD);
// 平面 BAD
ColoredVertex(ColorG, PointB);
ColoredVertex(ColorR, PointA);
ColoredVertex(ColorY, PointD);
glEnd();
glEndList();

glEnable(GL_DEPTH_TEST);
```

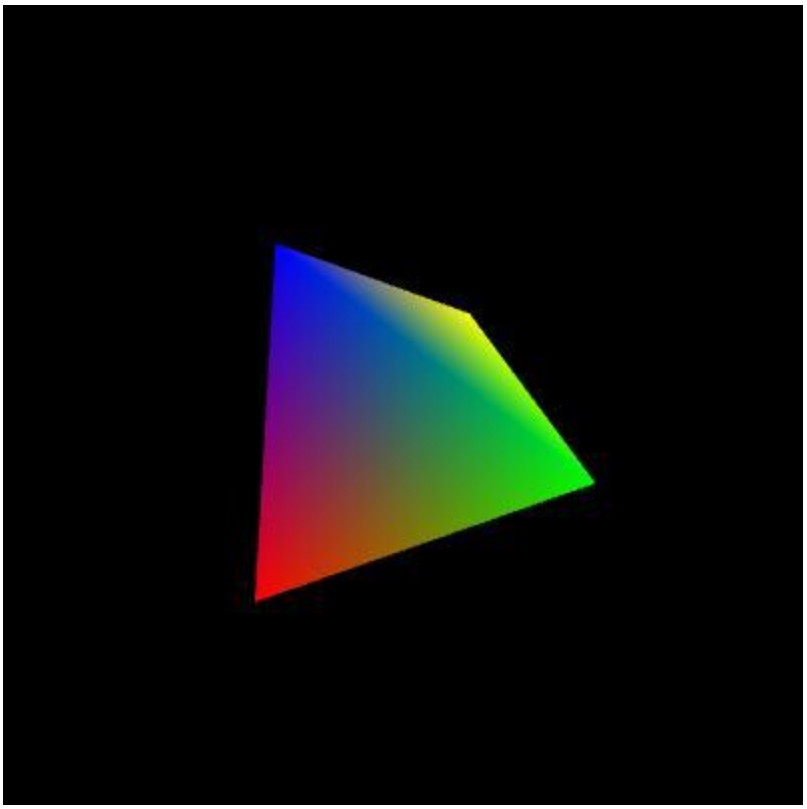
```
    }
    // 已经创建了显示列表，在每次绘制正四面体时将调用它
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    glRotatef(angle, 1, 0.5, 0);
    glCallList(list);
    glPopMatrix();
    glutSwapBuffers();
}

void myIdle(void)
{
    ++angle;
    if( angle >= 360.0f )
        angle = 0.0f;
    myDisplay();
}

int main(int argc, char* argv[])
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutInitWindowPosition(200, 200);
    glutInitWindowSize(WIDTH, HEIGHT);
    glutCreateWindow("OpenGL 窗口");
    glutDisplayFunc(&myDisplay);
    glutIdleFunc(&myIdle);
    glutMainLoop();
    return 0;
}
```

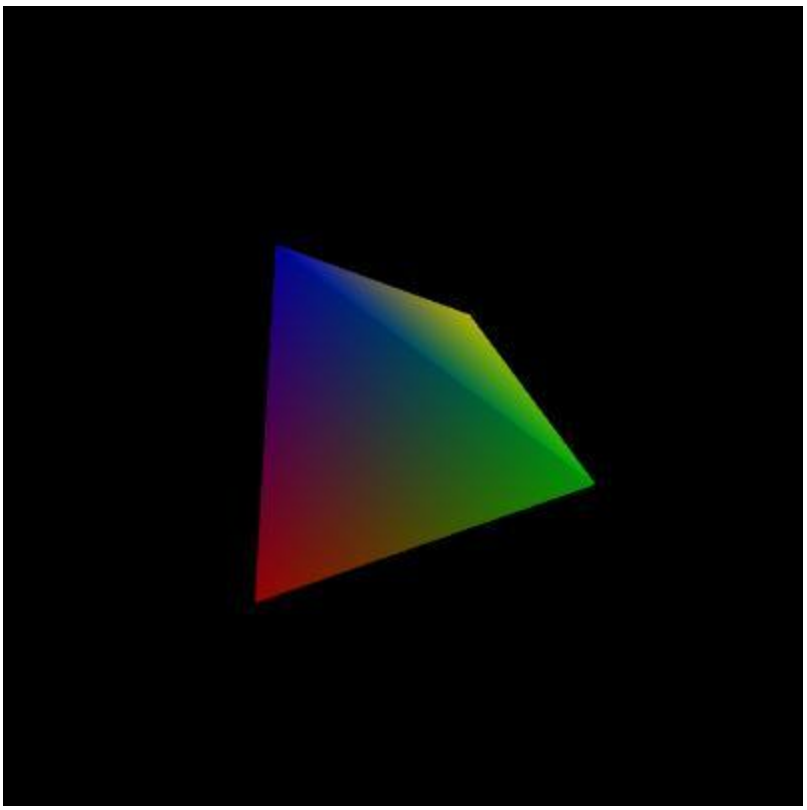
在程序中，我们将绘制正四面体的 OpenGL 函数装到了一个显示列表中，但是，关于旋转的操作却在显示列表之外进行。这是因为如果把旋转的操作也装入到显示列表，则每次旋转的角度都是一样的，不会随着 angle 的值的变化而变化，于是就不能表现出动态的旋转效果了。

程序运行时，可能感觉到画面的立体感不足，这主要是因为没有使用光照的缘故。如果将 glColor3fv 函数去掉，改为设置各种材质，然后开启光照效果，则可以产生更好的立体感。大家可以自己试着使用光照效果，唯一需要注意的地方就是法线向量的计算。由于这里的正四面体四个顶点坐标选取得比较特殊，使得正四面体的中心坐标正好是(0, 0, 0)，因此，每三个顶点坐标的平均值正好就是这三个顶点所组成的平面的法线向量的值。

```
void setNormal(GLfloat* Point1, GLfloat* Point2, GLfloat* Point3)
{
   GLfloat normal[3];
   int i;
   for(i=0; i<3; ++i)
      normal[i] = (Point1[i]+Point2[i]+Point3[i]) / 3;
   glNormal3fv(normal);
}
```

限于篇幅，这里就不给出完整的程序了。不过，大家可以自行尝试，看看使用光照后效果有何种改观。尤其是注意四面体各个表面交界的位置，在未使用光照前，几乎看不清轮廓，在使用光照后，可比较容易的区分各个平面，因此立体感得到加强。（见图 1，图 2）当然了，这样的效果还不够。如果在各表面的交界处设置很多细小的平面，进行平滑处理，则光照后的效果将更真实。但这已经远离本课的内容了。


图一


图二

小结

本课介绍了显示列表的知识和简单的应用。

可以把各种 OpenGL 函数调用的动作装到显示列表中，以后调用显示列表，就相当于调用了其中的 OpenGL 函数。显示列表中除了存放对 OpenGL 函数的调用外，不会存放其它内容。

使用显示列表的过程是：分配一个未使用的显示列表编号，把 OpenGL 函数调用装入显示列表，调用显示列表，销毁显示列表。

使用显示列表有可能带来程序运行速度的提升，但是这种提升并不一定会很明显。显示列表本身也存在一定的开销。

把绘制固定的物体的 OpenGL 函数放到一个显示列表中，是一种不错的编程思路。本课最后的例子中使用了这种思路。

9. 第九课：

今天介绍关于 OpenGL 混合的基本知识。混合是一种常用的技巧，通常可以用来实现半透明。但其实它也是十分灵活的，你可以通过不同的设置得到不同的混合结果，产生一些有趣或者奇怪的图象。

混合是什么呢？混合就是把两种颜色混在一起。具体一点，就是把某一像素位置原来的颜色和将要画上去的颜色，通过某种方式混在一起，从而实现特殊的效果。

假设我们需要绘制这样一个场景：透过红色的玻璃去看绿色的物体，那么可以先绘制绿色的物体，再绘制红色玻璃。在绘制红色玻璃的时候，利用“混合”功能，把将要绘制上去的红色和原来的绿色进行混合，于是得到一种新的颜色，看上去就好像玻璃是半透明的。

要使用 OpenGL 的混合功能，只需要调用：glEnable(GL_BLEND);即可。

要关闭 OpenGL 的混合功能，只需要调用：glDisable(GL_BLEND);即可。

注意：只有在 RGBA 模式下，才可以使用混合功能，颜色索引模式下是无法使用混合功能的。

一、 源因子和目标因子

前面我们已经提到，混合需要把原来的颜色和将要画上去的颜色找出来，经过某种方式处理后得到一种新的颜色。这里把将要画上去的颜色称为“源颜色”，把原来的颜色称为“目标颜色”。

OpenGL 会把源颜色和目标颜色各自取出，并乘以一个系数（源颜色乘以的系数称为“源因子”，目标颜色乘以的系数称为“目标因子”），然后相加，这样就得到了新的颜色。（也可以不是相加，新版本的 OpenGL 可以设置运算方式，包括加、减、取两者中较大的、取两者中较小的、逻辑运算等，但我们这里为了简单起见，不讨论这个了）

下面用数学公式来表达一下这个运算方式。假设源颜色的四个分量（指红色，绿色，蓝色，alpha 值）是(Rs, Gs, Bs, As)，目标颜色的四个分量是(Rd, Gd, Bd, Ad)，又设源因子为(Sr, Sg, Sb, Sa)，目标因子为(Dr, Dg, Db, Da)。则混合产生的新颜色可以表示为：

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

当然了，如果颜色的某一分量超过了 1.0，则它会被自动截取为 1.0，不需要考虑越界的问题。

源因子和目标因子是可以通过 glBlendFunc 函数来进行设置的。glBlendFunc 有两个参数，前者表示源因子，后者表示目标因子。这两个参数可以是多种值，下面介绍比较常用的几种。

GL_ZERO： 表示使用 0.0 作为因子，实际上相当于不使用这种颜色参与混合运算。

GL_ONE： 表示使用 1.0 作为因子，实际上相当于完全的使用了这种颜色参与混合运算。

GL_SRC_ALPHA：表示使用源颜色的 alpha 值来作为因子。

GL_DST_ALPHA：表示使用目标颜色的 alpha 值来作为因子。

GL_ONE_MINUS_SRC_ALPHA：表示用 1.0 减去源颜色的 alpha 值来作为因子。

GL_ONE_MINUS_DST_ALPHA：表示用 1.0 减去目标颜色的 alpha 值来作为因子。

除此以外，还有 GL_SRC_COLOR（把源颜色的四个分量分别作为因子的四个分量）、GL_ONE_MINUS_SRC_COLOR、GL_DST_COLOR、GL_ONE_MINUS_DST_COLOR 等，前两个在 OpenGL 旧版本中只能用于设置目标因子，后两个在 OpenGL 旧版本中只能用于设置源因子。新版本的 OpenGL 则没有这个限制，并且支持新的 GL_CONST_COLOR（设定一种常数颜色，将其四个分量分别作为因子的四个分量）、GL_ONE_MINUS_CONST_COLOR、GL_CONST_ALPHA、GL_ONE_MINUS_CONST_ALPHA。另外还有 GL_SRC_ALPHA_SATURATE。新版本的 OpenGL 还允许颜色的 alpha 值和 RGB 值采用不同的混合因子。但这些都不是我们现在所需要了解的。毕竟这还是入门教材，不需要整得太复杂~

举例来说：

如果设置了 glBlendFunc(GL_ONE, GL_ZERO);，则表示完全使用源颜色，完全不使用目标颜色，因此画面效果和不使用混合的时候一致（当然效率可能会低一点点）。如果没有设置源因子和目标因子，则默认情况就是这样的设置。

如果设置了 glBlendFunc(GL_ZERO, GL_ONE);，则表示完全不使用源颜色，因此无论你想画什么，最后都不会被画上去了。（但这并不是说这样设置就没有用，有些时候可能有特殊用途）

如果设置了 glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);，则表示源颜色乘以自身的 alpha 值，目标颜色乘以 1.0 减去源颜色的 alpha 值，这样一来，源颜色的 alpha 值越大，则产生的新颜色中源颜色所占比例就越大，而目标颜色所占比例则减小。这种情况下，我们可以简单的将源颜色的 alpha 值理解为“不透明度”。这也是混合时最常用的方式。

如果设置了 glBlendFunc(GL_ONE, GL_ONE);，则表示完全使用源颜色和目标颜色，最终的颜色实际上就是两种颜色的简单相加。例如红色(1, 0, 0)和绿色(0, 1, 0)相加得到(1, 1, 0)，结果为黄色。

注意：

所谓源颜色和目标颜色，是跟绘制的顺序有关的。假如先绘制了一个红色的物体，再在其上绘制绿色的物体。则绿色是源颜色，红色是目标颜色。如果顺序反过来，则红色就是源颜色，绿色才是目标颜色。在绘制时，应该注意顺序，使得绘制的源颜色与设置的源因子对应，目标颜色与设置的目标因子对应。不要被混乱的顺序搞晕了。

二、二维图形混合举例

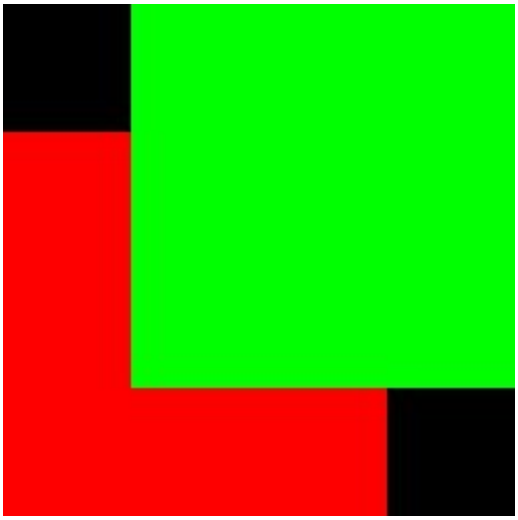
下面看一个简单的例子，实现将两种不同的颜色混合在一起。为了便于观察，我们绘制两个矩形：glRectf(-1, -1, 0.5, 0.5);glRectf(-0.5, -0.5, 1, 1);，这两个矩形有一个重叠的区域，便于我们观察混合的效果。

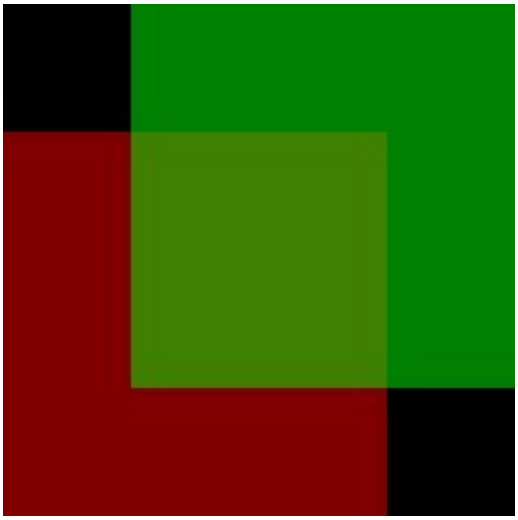
先来看看使用 glBlendFunc(GL_ONE, GL_ZERO);的，它的结果与不使用混合时相同。

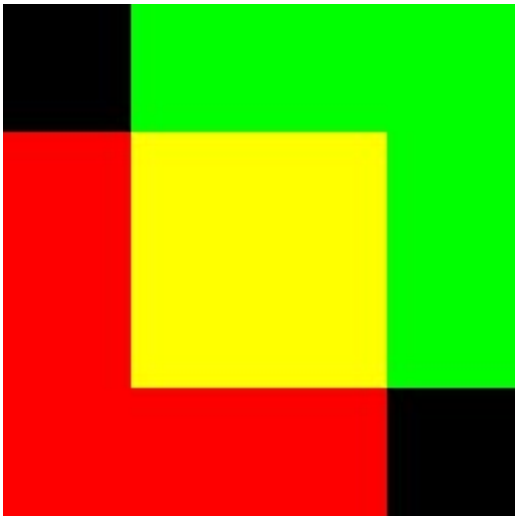
```
void myDisplay(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    glEnable(GL_BLEND);
    glBlendFunc(GL_ONE, GL_ZERO);
```

```
    glColor4f(1, 0, 0, 0.5);

    glRectf(-1, -1, 0.5, 0.5);

    glColor4f(0, 1, 0, 0.5);

    glRectf(-0.5, -0.5, 1, 1);


    glutSwapBuffers();

}
```

尝试把glBlendFunc的参数修改为glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA); 以及glBlendFunc(GL_ONE, GL_ONE);，观察效果。第一种情况下，效果与没有使用混合时相同，后绘制的图形会覆盖先绘制的图形。第二种情况下，alpha 被当作“不透明度”，由于被设置为 0.5，所以两个矩形看上去都是半透明的，乃至于看到黑色背景。第三种是将颜色相加，红色和绿色相加得到黄色。







二、 实现三维混合

也许你迫不及待的想要绘制一个三维的带有半透明物体的场景了。但是现在恐怕还不行，还有一点是在进行三维场景的混合时必须注意的，那就是深度缓冲。

深度缓冲是这样一段数据，它记录了每一个像素距离观察者有多近。在启用深度缓冲测试的情况下，如果将要绘制的像素比原来的像素更近，则像素将被绘制。否则，像素就会被忽略掉，不进行绘制。这在绘制不透明的物体时非常有用——不管是先绘制近的物体再绘制远的物体，还是先绘制远的物体再绘制近的物体，或者干脆以混乱的顺序进行绘制，最后的显示结果总是近的物体遮住远的物体。

然而在你需要实现半透明效果时，发现一切都不是那么美好了。如果你绘制了一个近距离的半透明物体，则它在深度缓冲区内保留了一些信息，使得远处的物体将无法再被绘制出来。虽然半透明的物体仍然半透明，但透过它看到的却不是正确的内容了。

要解决以上问题，需要在绘制半透明物体时将深度缓冲区设置为只读，这样一来，虽然半透明物体被绘制上去了，深度缓冲区还保持在原来的状态。如果再有一个物体出现在半透明物体之后，在不透明物体之前，则它也可以被绘制（因为此时深度缓冲区中记录的是那个不透明物体的深度）。以后再要绘制不透明物体时，只需要再将深度缓冲区设置为可读可写的形式即可。嗯？你问我怎么绘制一个一部分半透明一部分不透明的物体？这个好办，只需要把物体分为两个部分，一部分全是半透明的，一部分全是不透明的，分别绘制就可以了。

即使使用了以上技巧，我们仍然不能随心所欲的按照混乱顺序来进行绘制。必须是先绘制不透明的物体，然后绘制透明的物体。否则，假设背景为蓝色，近处一块红色玻璃，中间一个绿色物体。如果先绘制红色半透明玻璃的话，它先和蓝色背景进行混合，则以后绘制中间的绿色物体时，想单独与红色玻璃混合已经不能实现了。

总结起来，绘制顺序就是：首先绘制所有不透明的物体。如果两个物体都是不透明的，则谁先谁后都没有关系。然后，将深度缓冲区设置为只读。接下来，绘制所有半透明的物体。如果两个物体都是半透明的，则谁先谁后只需要根据自己的意愿（注意了，先绘制的将成为“目标颜色”，后绘制的将成为“源颜色”，所以绘制的顺序将会对结果造成一些影响）。最后，将深度缓冲区设置为可读可写形式。

调用 glDepthMask(GL_FALSE); 可将深度缓冲区设置为只读形式。调用 glDepthMask(GL_TRUE);可将深度缓冲区设置为可读可写形式。

一些网上的教程，包括大名鼎鼎的 NeHe 教程，都在使用三维混合时直接将深度缓冲区禁用，即调用 glDisable(GL_DEPTH_TEST);。这样做并不正确。如果先绘制一个不透明

的物体，再在其背后绘制半透明物体，本来后面的半透明物体将不会被显示（被不透明的物体遮住了），但如果禁用深度缓冲，则它仍然将会显示，并进行混合。NeHe 提到某些显卡在使用 glDepthMask 函数时可能存在一些问题，但可能是由于我的阅历有限，并没有发现这样的情况。

那么，实际的演示一下吧。我们来绘制一些半透明和不透明的球体。假设有三个球体，一个红色不透明的，一个绿色半透明的，一个蓝色半透明的。红色最远，绿色在中间，蓝色最近。根据前面所讲述的内容，红色不透明球体必须首先绘制，而绿色和蓝色则可以随意修改顺序。这里为了演示不注意设置深度缓冲的危害，我们故意先绘制最近的蓝色球体，再绘制绿色球体。

为了让这些球体有一点立体感，我们使用光照。在(1, 1, -1)处设置一个白色的光源。代码如下：

```
void setLight(void)
{
   static const GLfloat light_position[] = {1.0f, 1.0f, -1.0f, 1.0f};
   static const GLfloat light_ambient[]  = {0.2f, 0.2f, 0.2f, 1.0f};
   static const GLfloat light_diffuse[]  = {1.0f, 1.0f, 1.0f, 1.0f};
   static const GLfloat light_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};

   glLightfv(GL_LIGHT0, GL_POSITION, light_position);
   glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
   glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
   glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

   glEnable(GL_LIGHT0);
   glEnable(GL_LIGHTING);
   glEnable(GL_DEPTH_TEST);
}
```

每一个球体颜色不同。所以它们的材质也都不同。这里用一个函数来设置材质。

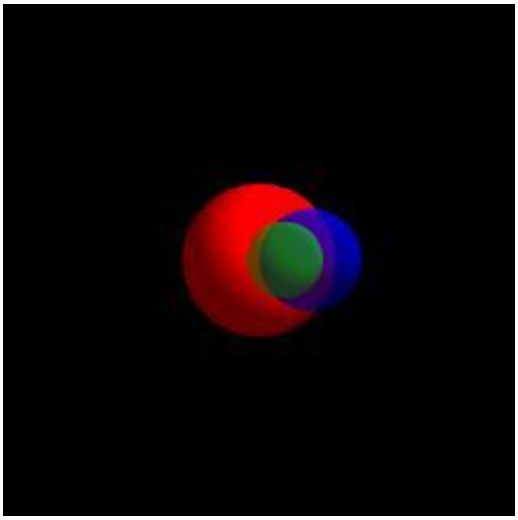
```
void setMatirial(const GLfloat mat_diffuse[4], GLfloat mat_shininess)
{
   static const GLfloat mat_specular[] = {0.0f, 0.0f, 0.0f, 1.0f};
   static const GLfloat mat_emission[] = {0.0f, 0.0f, 0.0f, 1.0f};

   glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, mat_diffuse);
   glMaterialfv(GL_FRONT, GL_SPECULAR,  mat_specular);
   glMaterialfv(GL_FRONT, GL_EMISSION,  mat_emission);
   glMaterialf (GL_FRONT, GL_SHININESS, mat_shininess);
}
```

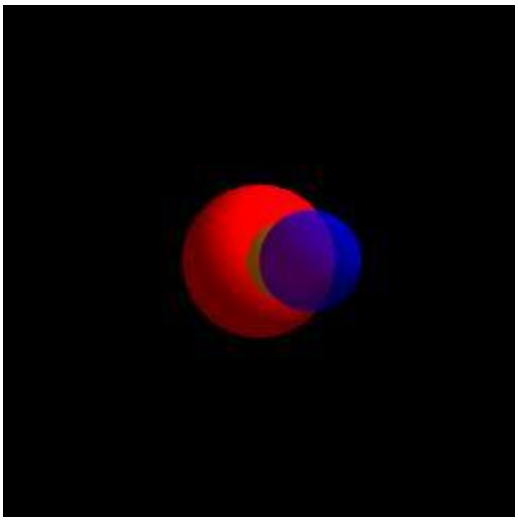
有了这两个函数，我们就可以根据前面的知识写出整个程序代码了。这里只给出了绘制的部分，其它部分大家可以自行完成。

```
void myDisplay(void)
{
   // 定义一些材质颜色
   const static GLfloat red_color[] = {1.0f, 0.0f, 0.0f, 1.0f};
   const static GLfloat green_color[] = {0.0f, 1.0f, 0.0f, 0.3333f};
   const static GLfloat blue_color[] = {0.0f, 0.0f, 1.0f, 0.5f};
```

```
    // 清除屏幕
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // 启动混合并设置混合因子
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);

    // 设置光源
    setLight();

    // 以(0, 0, 0.5)为中心，绘制一个半径为.3 的不透明红色球体（离观察者最远）
    setMatirial(red_color, 30.0);
    glPushMatrix();
    glTranslatef(0.0f, 0.0f, 0.5f);
    glutSolidSphere(0.3, 30, 30);
    glPopMatrix();

    // 下面将绘制半透明物体了，因此将深度缓冲设置为只读
    glDepthMask(GL_FALSE);

    // 以(0.2, 0, -0.5)为中心，绘制一个半径为.2 的半透明蓝色球体（离观察者最近）
    setMatirial(blue_color, 30.0);
    glPushMatrix();
    glTranslatef(0.2f, 0.0f, -0.5f);
    glutSolidSphere(0.2, 30, 30);
    glPopMatrix();

    // 以(0.1, 0, 0)为中心，绘制一个半径为.15 的半透明绿色球体（在前两个球体之间）
    setMatirial(green_color, 30.0);
    glPushMatrix();
    glTranslatef(0.1, 0, 0);
    glutSolidSphere(0.15, 30, 30);
    glPopMatrix();

    // 完成半透明物体的绘制，将深度缓冲区恢复为可读可写的形式
    glDepthMask(GL_TRUE);

    glutSwapBuffers();
}
```

大家也可以将上面两处 glDepthMask 删去，结果会看到最近的蓝色球虽然是半透明的，但它的背后直接就是红色球了，中间的绿色球没有被正确绘制。





小结：

本课介绍了 OpenGL 混合功能的相关知识。

混合就是在绘制时，不是直接把新的颜色覆盖在原来旧的颜色上，而是将新的颜色与旧的颜色经过一定的运算，从而产生新的颜色。新的颜色称为源颜色，原来旧的颜色称为目标颜色。传统意义上的混合，是将源颜色乘以源因子，目标颜色乘以目标因子，然后相加。

源因子和目标因子是可以设置的。源因子和目标因子设置的不同直接导致混合结果的不同。将源颜色的 alpha 值作为源因子，用 1.0 减去源颜色 alpha 值作为目标因子，是一种常用的方式。这时候，源颜色的 alpha 值相当于“不透明度”的作用。利用这一特点可以绘制出一些半透明的物体。

在进行混合时，绘制的顺序十分重要。因为在绘制时，正要绘制上去的是源颜色，原来存在的是目标颜色，因此先绘制的物体就成为目标颜色，后来绘制的则成为源颜色。绘制的顺序要考虑清楚，将目标颜色和设置的目标因子相对应，源颜色和设置的源因子相对应。

在进行三维混合时，不仅要考虑源因子和目标因子，还应该考虑深度缓冲区。必须先绘制所有不透明的物体，再绘制半透明的物体。在绘制半透明物体时前，还需要将深度缓冲区设置为只读形式，否则可能出现画面错误。

10.第十课：

第十课、第十一课中的一个函数中的 bug：在 grab 函数中，应该在最开头加上一句 glReadBuffer(GL_FRONT);以保证读取到的内容正好就是显示的内容。

学过多媒体技术的朋友可能知道，计算机保存图象的方法通常有两种：一是“矢量图”，一是“像素图”。矢量图保存了图象中每一几何物体的位置、形状、大小等信息，在显示图象时，根据这些信息计算得到完整的图象。“像素图”是将完整的图象纵横分为若干的行、列，这些行列使得图象被分割为很细小的分块，每一分块称为像素，保存每一像素的颜色也就保存了整个图象。

这两种方法各有优缺点。“矢量图”在图象进行放大、缩小时很方便，不会失真，但如果图象很复杂，那么就需要用非常多的几何体，数据量和运算量都很庞大。“像素图”无论图象多么复杂，数据量和运算量都不会增加，但在进行放大、缩小等操作时，会产生失真的情况。

前面我们曾介绍了如何使用 OpenGL 来绘制几何体，我们通过重复的绘制许多几何体，可以绘制出一幅矢量图。那么，应该如何绘制像素图呢？这就是我们今天要学习的内容了。

1、BMP 文件格式简单介绍

BMP 文件是一种像素文件，它保存了一幅图象中所有的像素。这种文件格式可以保存单色位图、16 色或 256 色索引模式像素图、24 位真彩色图象，每种模式种单一像素的大小分别为 1/8 字节，1/2 字节，1 字节和 3 字节。目前最常见的是 256 色 BMP 和 24 位色 BMP。这种文件格式还定义了像素保存的几种方法，包括不压缩、RLE 压缩等。常见的 BMP 文件大多是不压缩的。

这里为了简单起见，我们仅讨论 24 位色、不使用压缩的 BMP。（如果你使用 Windows 自带的画图程序，很容易绘制出一个符合以上要求的 BMP）

Windows 所使用的 BMP 文件，在开始处有一个文件头，大小为 54 字节。保存了包括文件格式标识、颜色数、图象大小、压缩方式等信息，因为我们仅讨论 24 位色不压缩的 BMP，所以文件头中的信息基本不需要注意，只有“大小”这一项对我们比较有用。图象的宽度和高度都是一个 32 位整数，在文件中的地址分别为 0x0012 和 0x0016，于是我们可以使用以下代码来读取图象的大小信息：

```
GLint width, height; // 使用 OpenGL 的 GLint 类型，它是 32 位的。
                     // 而 C 语言本身的 int 则不一定是 32 位的。
FILE* pFile;
// 在这里进行“打开文件”的操作
fseek(pFile, 0x0012, SEEK_SET);     // 移动到 0x0012 位置
fread(&width, sizeof(width), 1, pFile); // 读取宽度
fseek(pFile, 0x0016, SEEK_SET);     // 移动到 0x0016 位置
                                    // 由于上一句执行后本就应该在 0x0016 位置
                                    // 所以这一句可省略
fread(&height, sizeof(height), 1, pFile); // 读取高度
```

54 个字节以后，如果是 16 色或 256 色 BMP，则还有一个颜色表，但 24 位色 BMP 没有这个，我们这里不考虑。接下来就是实际的像素数据了。24 位色的 BMP 文件中，每三个字节表示一个像素的颜色。

注意，OpenGL 通常使用 RGB 来表示颜色，但 BMP 文件则采用 BGR，就是说，顺序被反过来了。另外需要注意的地方是：像素的数据量并不一定完全等于图象的高度乘以宽度乘以每一像素的字节数，而是可能略大于这个值。原因是 BMP 文件采用了一种“对齐”的机制，每一行像素数据的长度若不是 4 的倍数，则填充一些数据使它是 4 的倍数。这样一来，一个 17*15 的 24 位 BMP 大小就应该是 834 字节（每行 17 个像素，有 51 字节，补充为 52 字节，乘以 15 得到像素数据总长度 780，再加上文件开始的 54 字节，得到 834 字节）。分配内存时，一定要小心，不能直接使用“图象的高度乘以宽度乘以每一像素的字节数”来计算分配空间的长度，否则有可能导致分配的内存空

间长度不足，造成越界访问，带来各种严重后果。
一个很简单的计算数据长度的方法如下：

```
int LineLength, TotalLength;
LineLength = ImageWidth * BytesPerPixel; // 每行数据长度大致为图象宽度乘以
                                         // 每像素的字节数
while( LineLength % 4 != 0 )             // 修正 LineLength 使其为 4 的倍数
  ++LineLenth;
TotalLength = LineLength * ImageHeight; // 数据总长 = 每行长度 * 图象高度
```

这并不是效率最高的方法，但由于这个修正本身运算量并不大，使用频率也不高，我们就不需要再考虑更快的方法了。

2、简单的 OpenGL 像素操作

OpenGL 提供了简洁的函数来操作像素：

glReadPixels：读取一些像素。当前可以简单理解为“把已经绘制好的像素（它可能已经被保存到显卡的显存中）读取到内存”。

glDrawPixels：绘制一些像素。当前可以简单理解为“把内存中一些数据作为像素数据，进行绘制”。

glCopyPixels：复制一些像素。当前可以简单理解为“把已经绘制好的像素从一个位置复制到另一个位置”。虽然从功能上看，好象等价于先读取像素再绘制像素，但实际上它不需要把已经绘制的像素（它可能已经被保存到显卡的显存中）转换为内存数据，然后再由内存数据进行重新的绘制，所以要比先读取后绘制快很多。

这三个函数可以完成简单的像素读取、绘制和复制任务，但实际上也可以完成更复杂的任务。当前，我们仅讨论一些简单的应用。由于这几个函数的参数数目比较多，下面我们分别介绍。

3、glReadPixels 的用法和举例

3.1 函数的参数说明

该函数总共有七个参数。前四个参数可以得到一个矩形，该矩形所包括的像素都会被读取出来。（第一、二个参数表示了矩形的左下角横、纵坐标，坐标以窗口最左下角为零，最右上角为最大值；第三、四个参数表示了矩形的宽度和高度）

第五个参数表示读取的内容，例如：GL_RGB 就会依次读取像素的红、绿、蓝三种数据，GL_RGBA 则会依次读取像素的红、绿、蓝、alpha 四种数据，GL_RED 则只读取像素的红色数据（类似的还有 GL_GREEN，GL_BLUE，以及 GL_ALPHA）。如果采用的不是 RGBA 颜色模式，而是采用颜色索引模式，则也可以使用 GL_COLOR_INDEX 来读取像素的颜色索引。目前仅需要知道这些，但实际上还可以读取其它内容，例如深度缓冲区的深度数据等。

第六个参数表示读取的内容保存到内存时所使用的格式，例如：GL_UNSIGNED_BYTE 会把各种数据保存为 GLubyte，GL_FLOAT 会把各种数据保存为 GLfloat 等。

第七个参数表示一个指针，像素数据被读取后，将被保存到这个指针所表示的地址。注意，需要保证该地址有足够的可以使用的空间，以容纳读取的像素数据。例如一幅大小为 256*256 的图象，如果读取其 RGB 数据，且每一数据被保存为 GLubyte，总大小就是：256*256*3 = 196608 字节，即 192 千字节。如果是读取 RGBA 数据，则总大小就是 256*256*4 = 262144 字节，即 256 千字节。

注意：glReadPixels 实际上是从缓冲区中读取数据，如果使用了双缓冲区，则默认是从正在显示的缓冲（即前缓冲）中读取，而绘制工作是默认绘制到后缓冲区的。因此，如果需要读取已经绘制好的像素，往往需要先交换前后缓冲。

再看前面提到的 BMP 文件中两个需要注意的地方：

3.2 解决 OpenGL 常用的 RGB 像素数据与 BMP 文件的 BGR 像素数据顺序不一致问题

可以使用一些代码交换每个像素的第一字节和第三字节，使得 RGB 的数据变成 BGR 的数据。当然也可以使用另外的方式解决问题：新版本的 OpenGL 除了可以使用 GL_RGB 读取像素的红、绿、蓝数据外，也可以使用 GL_BGR 按照相反的顺序依次读取像素的蓝、绿、红数据，这样就与 BMP 文件格式相吻合了。即使你的 gl/gl.h 头文件中没有定义这个 GL_BGR，也没有关系，可以尝试使用 GL_BGR_EXT。虽然有的 OpenGL 实现（尤其是旧版本的实现）并不能使用 GL_BGR_EXT，但我所知道的 Windows 环境下各种 OpenGL 实现都对 GL_BGR 提供了支持，毕竟 Windows 中各种表示颜色的数据几乎都是使用 BGR 的顺序，而非 RGB 的顺序。这可能与 IBM-PC 的硬件设计有关。

3.3 消除 BMP 文件中“对齐”带来的影响

实际上 OpenGL 也支持使用了这种“对齐”方式的像素数据。只要通过 glPixelStore 修改“像素保存时对齐的方式”就可以了。像这样：

```
int alignment = 4;
glPixelStorei(GL_UNPACK_ALIGNMENT, alignment);
```

第一个参数表示“设置像素的对齐值”，第二个参数表示实际设置为多少。这里像素可以单字节对齐（实际上就是不使用对齐）、双字节对齐（如果长度为奇数，则再补一个字节）、四字节对齐（如果长度不是四的倍数，则补为四的倍数）、八字节对齐。分别对应 alignment 的值为 1, 2, 4, 8。实际上，默认的值是 4，正好与 BMP 文件的对齐方式相吻合。

glPixelStorei 也可以用于设置其它各种参数。但我们这里并不需要深入讨论了。

现在，我们已经可以把屏幕上的像素读取到内存了，如果需要的话，我们还可以将内存中的数据保存到文件。正确的对照 BMP 文件格式，我们的程序就可以把屏幕中的图象保存为 BMP 文件，达到屏幕截图的效果。

我们并没有详细介绍 BMP 文件开头的 54 个字节的所有内容，不过这无伤大雅。从一个正确的 BMP 文件中读取前 54 个字节，修改其中的宽度和高度信息，就可以得到新的文件头了。假设我们先建立一个 1*1 大小的 24 位色 BMP，文件名为 dummy.bmp，又假设新的 BMP 文件名称为 grab.bmp。则可以编写如下代码：

```
FILE* pOriginFile = fopen("dummy.bmp", "rb");
FILE* pGrabFile = fopen("grab.bmp", "wb");
char  BMP_Header[54];
GLint width, height;

/* 先在这里设置好图象的宽度和高度，即 width 和 height 的值，并计算像素的总长度 */

// 读取 dummy.bmp 中的头 54 个字节到数组
fread(BMP_Header, sizeof(BMP_Header), 1, pOriginFile);
// 把数组内容写入到新的 BMP 文件
fwrite(BMP_Header, sizeof(BMP_Header), 1, pGrabFile);

// 修改其中的大小信息
```

```
        fseek(pGrabFile, 0x0012, SEEK_SET);
        fwrite(&width, sizeof(width), 1, pGrabFile);
        fwrite(&height, sizeof(height), 1, pGrabFile);

        // 移动到文件末尾，开始写入像素数据
        fseek(pGrabFile, 0, SEEK_END);

        /* 在这里写入像素数据到文件 */

        fclose(pOriginFile);
        fclose(pGrabFile);
```

我们给出完整的代码，演示如何把整个窗口的图象抓取出来并保存为 BMP 文件。

```
#define WindowWidth  400
#define WindowHeight 400


#include <stdio.h>
#include <stdlib.h>


/* 函数 grab
 * 抓取窗口中的像素
 * 假设窗口宽度为 WindowWidth，高度为 WindowHeight
 */
#define BMP_Header_Length 54
void grab(void)
{
   FILE*    pDummyFile;
   FILE*    pWritingFile;
   GLubyte* pPixelData;
   GLubyte  BMP_Header[BMP_Header_Length];
   GLint    i, j;
   GLint    PixelDataLength;


   // 计算像素数据的实际长度
```

```
i = WindowWidth * 3;  // 得到每一行的像素数据长度
while( i%4 != 0 )     // 补充数据，直到 i 是的倍数
   ++i;               // 本来还有更快的算法，
                      // 但这里仅追求直观，对速度没有太高要求
PixelDataLength = i * WindowHeight;

// 分配内存和打开文件
pPixelData = (GLubyte*)malloc(PixelDataLength);
if( pPixelData == 0 )
   exit(0);

pDummyFile = fopen("dummy.bmp", "rb");
if( pDummyFile == 0 )
   exit(0);

pWritingFile = fopen("grab.bmp", "wb");
if( pWritingFile == 0 )
   exit(0);

// 读取像素
glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
glReadPixels(0, 0, WindowWidth, WindowHeight,
   GL_BGR_EXT, GL_UNSIGNED_BYTE, pPixelData);

// 把 dummy.bmp 的文件头复制为新文件的文件头
fread(BMP_Header, sizeof(BMP_Header), 1, pDummyFile);
fwrite(BMP_Header, sizeof(BMP_Header), 1, pWritingFile);
fseek(pWritingFile, 0x0012, SEEK_SET);
i = WindowWidth;
j = WindowHeight;
```

```
    fwrite(&i, sizeof(i), 1, pWritingFile);
    fwrite(&j, sizeof(j), 1, pWritingFile);

    // 写入像素数据
    fseek(pWritingFile, 0, SEEK_END);
    fwrite(pPixelData, PixelDataLength, 1, pWritingFile);

    // 释放内存和关闭文件
    fclose(pDummyFile);
    fclose(pWritingFile);
    free(pPixelData);
}
```

把这段代码复制到以前任何课程的样例程序中，在绘制函数的最后调用 grab 函数，即可把图象内容保存为 BMP 文件了。（在我写这个教程的时候，不少地方都用这样的代码进行截图工作，这段代码一旦写好，运行起来是很方便的。）

4、glDrawPixels 的用法和举例

glDrawPixels 函数与 glReadPixels 函数相比，参数内容大致相同。它的第一、二、三、四个参数分别对应于 glReadPixels 函数的第三、四、五、六个参数，依次表示图象宽度、图象高度、像素数据内容、像素数据在内存中的格式。两个函数的最后一个参数也是对应的，glReadPixels 中表示像素读取后存放在内存中的位置，glDrawPixels 则表示用于绘制的像素数据在内存中的位置。

注意到 glDrawPixels 函数比 glReadPixels 函数少了两个参数，这两个参数在 glReadPixels 中分别是表示图象的起始位置。在 glDrawPixels 中，不必显式的指定绘制的位置，这是因为绘制的位置是由另一个函数 glRasterPos*来指定的。glRasterPos*函数的参数与 glVertex*类似，通过指定一个二维/三维/四维坐标，OpenGL 将自动计算出该坐标对应的屏幕位置，并把该位置作为绘制像素的起始位置。

很自然的，我们可以从 BMP 文件中读取像素数据，并使用 glDrawPixels 绘制到屏幕上。我们选择 Windows XP 默认的桌面背景 Bliss.bmp 作为绘制的内容（如果你使用的是 Windows XP 系统，很可能可以在硬盘中搜索到这个文件。当然你也可以使用其它 BMP 文件来代替，只要它是 24 位的 BMP 文件。注意需要修改代码开始部分的 FileName 的定义），先把该文件复制一份放到正确的位置，我们在程序开始时，就读取该文件，从而获得图象的大小后，根据该大小来创建合适的 OpenGL 窗口，并绘制像素。

绘制像素本来是很简单的过程，但是这个程序在骨架上与前面的各种示例程序稍有不同，所以我还是打算给出一份完整的代码。

```
#include <gl/glut.h>

#define FileName "Bliss.bmp"

static GLint    ImageWidth;
static GLint    ImageHeight;
static GLint    PixelLength;
static GLubyte* PixelData;

#include <stdio.h>
#include <stdlib.h>

void display(void)
{
   // 清除屏幕并不必要
   // 每次绘制时，画面都覆盖整个屏幕
   // 因此无论是否清除屏幕，结果都一样
   // glClear(GL_COLOR_BUFFER_BIT);

   // 绘制像素
   glDrawPixels(ImageWidth, ImageHeight,
      GL_BGR_EXT, GL_UNSIGNED_BYTE, PixelData);

   // 完成绘制
   glutSwapBuffers();
}
```

```
int main(int argc, char* argv[])
{
    // 打开文件
    FILE* pFile = fopen("Bliss.bmp", "rb");
    if( pFile == 0 )
        exit(0);

    // 读取图象的大小信息
    fseek(pFile, 0x0012, SEEK_SET);
    fread(&ImageWidth, sizeof(ImageWidth), 1, pFile);
    fread(&ImageHeight, sizeof(ImageHeight), 1, pFile);

    // 计算像素数据长度
    PixelLength = ImageWidth * 3;
    while( PixelLength % 4 != 0 )
        ++PixelLength;
    PixelLength *= ImageHeight;

    // 读取像素数据
    PixelData = (GLubyte*)malloc(PixelLength);
    if( PixelData == 0 )
        exit(0);

    fseek(pFile, 54, SEEK_SET);
    fread(PixelData, PixelLength, 1, pFile);

    // 关闭文件
    fclose(pFile);

    // 初始化 GLUT 并运行
```

```
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(ImageWidth, ImageHeight);
    glutCreateWindow(FileName);
    glutDisplayFunc(&display);
    glutMainLoop();

    // 释放内存
    // 实际上，glutMainLoop 函数永远不会返回，这里也永远不会到达
    // 这里写释放内存只是出于一种个人习惯
    // 不用担心内存无法释放。在程序结束时操作系统会自动回收所有内存
    free(PixelData);

    return 0;
}
```

这里仅仅是一个简单的显示 24 位 BMP 图象的程序，如果读者对 BMP 文件格式比较熟悉，也可以写出适用于各种 BMP 图象的显示程序，在像素处理时，它们所使用的方法是类似的。

OpenGL 在绘制像素之前，可以对像素进行若干处理。最常用的可能就是对整个像素图象进行放大/缩小。使用 glPixelZoom 来设置放大/缩小的系数，该函数有两个参数，分别是水平方向系数和垂直方向系数。例如设置 glPixelZoom(0.5f, 0.8f);则表示水平方向变为原来的 50%大小，而垂直方向变为原来的 80%大小。我们甚至可以使用负的系数，使得整个图象进行水平方向或垂直方向的翻转（默认像素从左绘制到右，但翻转后将从右绘制到左。默认像素从下绘制到上，但翻转后将从上绘制到下。因此，glRasterPos*函数设置的“开始位置”不一定就是矩形的左下角）。

5、glCopyPixels 的用法和举例

从效果上看，glCopyPixels 进行像素复制的操作，等价于把像素读取到内存，再从内存绘制到另一个区域，因此可以通过 glReadPixels 和 glDrawPixels 组合来实现复制像素的功能。然而我们知道，像素数据通常数据量很大，例如一幅 1024*768 的图象，如果使用 24 位 BGR 方式表示，则需要至少 1024*768*3 字节，即 2.25 兆字节。这么多的数据要进行一次读操作和一次写操作，并且因为在 glReadPixels 和 glDrawPixels 中设置的数据格式不同，很可能涉及到数据格式的转换。这对 CPU 无疑是一个不小的负担。使用 glCopyPixels 直接从像素数据复制出新的像素数据，避免了多余的数据的格式转换，并且也可能减少一些数据复制操

作（因为数据可能直接由显卡负责复制，不需要经过主内存），因此效率比较高。

glCopyPixels 函数也通过 glRasterPos*系列函数来设置绘制的位置，因为不需要涉及到主内存，所以不需要指定数据在内存中的格式，也不需要使用任何指针。

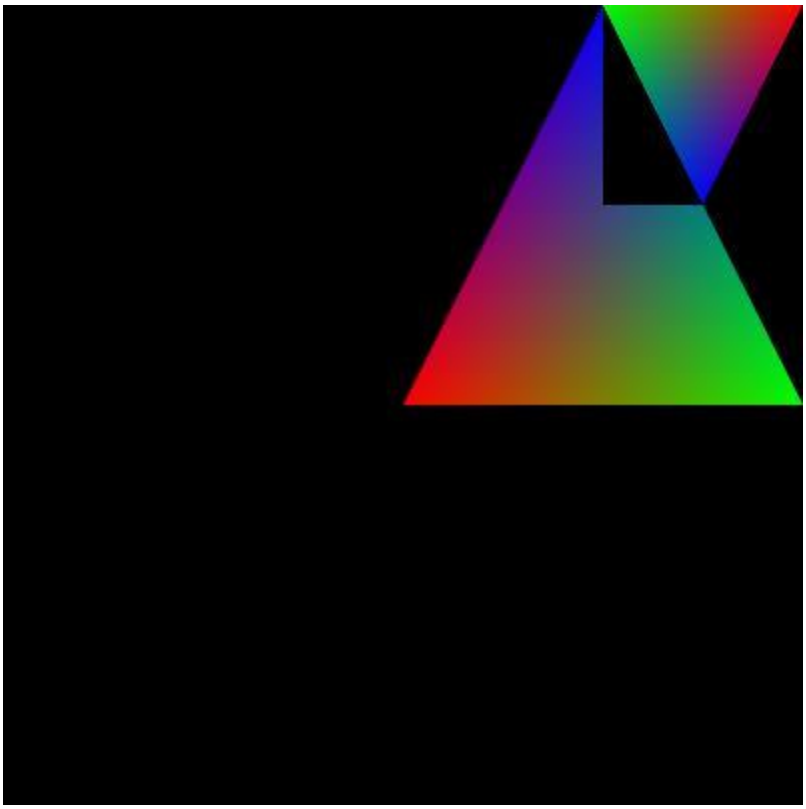
glCopyPixels 函数有五个参数，第一、二个参数表示复制像素来源的矩形的左下角坐标，第三、四个参数表示复制像素来源的举行的宽度和高度，第五个参数通常使用 GL_COLOR，表示复制像素的颜色，但也可以是 GL_DEPTH 或 GL_STENCIL，分别表示复制深度缓冲数据或模板缓冲数据。

值得一提的是，glDrawPixels 和 glReadPixels 中设置的各种操作，例如 glPixelZoom 等，在 glCopyPixels 函数中同样有效。

下面看一个简单的例子，绘制一个三角形后，复制像素，并同时进行水平和垂直方向的翻转，然后缩小为原来的一半，并绘制。绘制完毕后，调用前面的 grab 函数，将屏幕中所有内容保存为 grab.bmp。其中 WindowWidth 和 WindowHeight 是表示窗口宽度和高度的常量。

```
void display(void)
{
  // 清除屏幕
  glClear(GL_COLOR_BUFFER_BIT);

  // 绘制
  glBegin(GL_TRIANGLES);
    glColor3f(1.0f, 0.0f, 0.0f);   glVertex2f(0.0f, 0.0f);
    glColor3f(0.0f, 1.0f, 0.0f);   glVertex2f(1.0f, 0.0f);
    glColor3f(0.0f, 0.0f, 1.0f);   glVertex2f(0.5f, 1.0f);
  glEnd();
  glPixelZoom(-0.5f, -0.5f);
  glRasterPos2i(1, 1);
  glCopyPixels(WindowWidth/2, WindowHeight/2,
    WindowWidth/2, WindowHeight/2, GL_COLOR);

  // 完成绘制，并抓取图象保存为 BMP 文件
  glutSwapBuffers();
```

```
    grab();
}
```



小结：

本课结合 Windows 系统常见的 BMP 图象格式，简单介绍了 OpenGL 的像素处理功能。包括使用 glReadPixels 读取像素、glDrawPixels 绘制像素、glCopyPixels 复制像素。

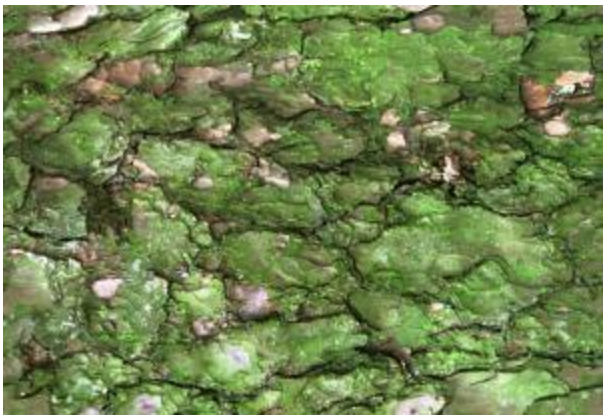
本课仅介绍了像素处理的一些简单应用，但相信大家已经可以体会到，围绕这三个像素处理函数，还存在一些“外围”函数，比如 glPixelStore*，glRasterPos*，以及 glPixelZoom 等。我们仅使用了这些函数的一少部分功能。

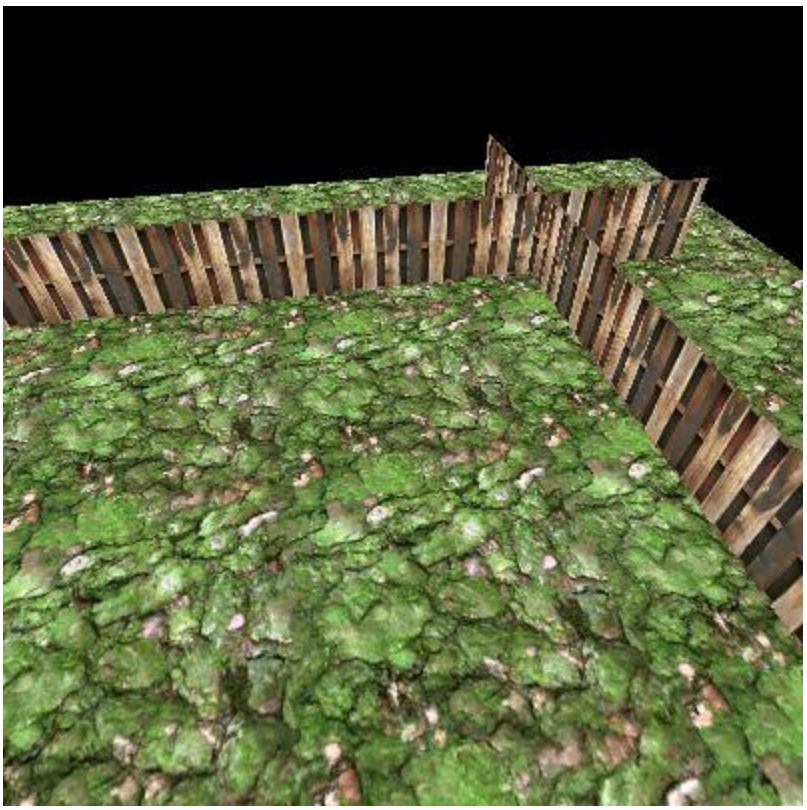
本课内容并不多，例子足够丰富，三个像素处理函数都有例子，大家可以结合例子来体会。

11. 第十一课:

我们在前一课中，学习了简单的像素操作，这意味着我们可以使用各种各样的 BMP 文件来丰富程序的显示效果，于是我们的 OpenGL 图形程序也不再像以前总是只显示几个多边形那样单调了。——但是这还不够。虽然我们可以将像素数据按照矩形进行缩小和放大，但是还不足以满足我们的要求。例如要将一幅世界地图绘制到一个球体表面，只使用 glPixelZoom 这样的函数来进行缩放显然是不够的。OpenGL 纹理映射功能支持将一些像素数据经过变换（即使是比较不规则的变换）将其附着到各种形状的多边形表面。纹理映射功能十分强大，利用它可以实现目前计算机动画中的大多数效果，但是它也很复杂，我们不可能一次性的完全讲解。这里的课程只是关于二维纹理的简单使用。但即使是这样，也会使我们的程序在显示效果上迈出一大步。

下面几张图片说明了纹理的效果。前两张是我们需要的纹理，后一张是我们使用纹理后，利用 OpenGL 所产生出的效果。





纹理的使用是非常复杂的。因此即使是入门教程，在编写时我也多次进行删改，很多东西都被精

简掉了，但本课的内容仍然较多，大家要有一点心理准备~

1、启用纹理和载入纹理

就像我们曾经学习过的 OpenGL 光照、混合等功能一样。在使用纹理前，必须启用它。OpenGL 支持一维纹理、二维纹理和三维纹理，这里我们仅介绍二维纹理。可以使用以下语句来启用和禁用二维纹理：

```
glEnable(GL_TEXTURE_2D); // 启用二维纹理
glDisable(GL_TEXTURE_2D); // 禁用二维纹理
```

使用纹理前，还必须载入纹理。利用 glTexImage2D 函数可以载入一个二维的纹理，该函数有多达九个参数（虽然某些参数我们可以暂时不去了解），现在分别说明如下：

第一个参数为指定的目标，在我们的入门教材中，这个参数将始终使用 GL_TEXTURE_2D。

第二个参数为“多重细节层次”，现在我们并不考虑多重纹理细节，因此这个参数设置为零。

第三个参数有两种用法。在 OpenGL 1.0，即最初的版本中，使用整数来表示颜色分量数目，例如：像素数据用 RGB 颜色表示，总共有红、绿、蓝三个值，因此参数设置为 3，而如果像素数据是用 RGBA 颜色表示，总共有红、绿、蓝、alpha 四个值，因此参数设置为 4。而在后来的版本中，可以直接使用 GL_RGB 或 GL_RGBA 来表示以上情况，显得更直观（并带来其它一些好处，这里暂时不提）。注意：虽然我们使用 Windows 的 BMP 文件作为纹理时，一般是蓝色的像素在最前，其真实的格式为 GL_BGR 而不是 GL_RGB，在数据的顺序上有所不同，但因为同样是红、绿、蓝三种颜色，因此这里仍然使用 GL_RGB。（如果使用 GL_BGR，OpenGL 将无法识别这个参数，造成错误）

第四、五个参数是二维纹理像素的宽度和高度。这里有一个很需要注意的地方：OpenGL 在以前的很多版本中，限制纹理的大小必须是 2 的整数次方，即纹理的宽度和高度只能是 16, 32, 64, 128, 256 等值，直到最近的新版本才取消了这个限制。而且，一些 OpenGL 实现（例如，某些 PC 机上板载显卡的驱动程序附带的 OpenGL）并没有支持到如此高的 OpenGL 版本。因此在使用纹理时要特别注意其大小。尽量使用大小为 2 的整数次方的纹理，当这个要求无法满足时，使用 gluScaleImage 函数把图象缩放至所指定的大小（在后面的例子中有用到）。另外，无论旧版本还是新版本，都限制了纹理大小的最大值，例如，某 OpenGL 实现可能要求纹理最大不能超过 1024*1024。可以使用如下的代码来获得 OpenGL 所支持的最大纹理：

```
GLint max;
glGetIntegerv(GL_MAX_TEXTURE_SIZE, &max);
```

这样 max 的值就是当前 OpenGL 实现中所支持的最大纹理。

在很长一段时间内，很多图形程序都喜欢使用 256*256 大小的纹理，不仅因为 256 是 2 的整数次方，也因为某些硬件可以使用 8 位的整数来表示纹理坐标，2 的 8 次方正好是 256，这一巧妙的组合为处理纹理坐标时的硬件优化创造了一些不错的条件。

第六个参数是纹理边框的大小，我们没有使用纹理边框，因此这里设置为零。

最后三个参数与 glDrawPixels 函数的最后三个参数的使用方法相同，其含义可以参考 glReadPixels 的最后三个参数。大家可以复习一下第 10 课的相关内容，这里不再重复。

举个例子，如果有一幅大小为 width*height，格式为 Windows 系统中使用最普遍的 24 位 BGR，保存在 pixels 中的像素图象。则把这样一幅图象载入为纹理可使用以下代码：

```
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_BGR_EXT, GL_UNSIGNED_BYTE, pixels);
```

> 注意，载入纹理的过程可能比较慢，原因是纹理数据通常比较大，例如一幅 512*512 的 BGR 格式的图象，大小为 0.75M。把这些像素数据从主内存传送到专门的图形硬件，这个过程中还可能需要把程序中所指定的像素格式转化为图形硬件所能识别的格式（或最能发挥图形硬件性能的格式），这些操作都需要较多时间。

2、纹理坐标

我们先来回忆一下之前学过的一点内容：

当我们绘制一个三角形时，只需要指定三个顶点的颜色。三角形中其它各点的颜色不需要我们指定，这些点的颜色是 OpenGL 自己通过计算得到的。

在我们学习 OpneGL 光照时，法线向量、材质的指定，都是只需要在顶点处指定一下就可以了，其它地方的法线向量和材质都是 OpenGL 自己通过计算去获得。

纹理的使用方法也与此类似。只要指定每一个顶点在纹理图象中所对应的像素位置，OpenGL 就会自动计算顶点以外的其它点在纹理图象中所对应的像素位置。

这听起来比较令人迷惑。我们可以这样类比一下：

在绘制一条线段时，我们设置其中一个端点为红色，另一个端点为绿色，则 OpenGL 会自动计算线段中其它各像素的颜色，如果是使用 glShadeMode(GL_SMOOTH);，则最终会形成一种渐变的效果（例如线段中点，就是红色和绿色的中间色）。

类似的，在绘制一条线段时，我们设置其中一个端点使用“纹理图象中最左下角的颜色”作为它的颜色，另一个端点使用“纹理图象中最右上角的颜色”作为它的颜色，则 OpenGL 会自动在纹理图象中选择合适位置

的颜色，填充到线段的各个像素（例如线段中点，可能就是选择纹理图象中央的那个像素的颜色）。

我们在类比时，使用了“纹理图象中最左下角的颜色”这种说法。但这种说法在很多时候不够精确，我们需要一种精确的方式来表示我们究竟使用纹理中的哪个像素。纹理坐标也就是因为这样的要求而产生的。以二维纹理为例，规定纹理最左下角的坐标为(0, 0)，最右上角的坐标为(1, 1)，于是纹理中的每一个像素的位置都可以用两个浮点数来表示（三维纹理会用三个浮点数表示，一维纹理则只用一个即可）。

使用 glTexCoord*系列函数来指定纹理坐标。这些函数的用法与使用 glVertex*系列函数来指定顶点坐标十分相似。例如：glTexCoord2f(0.0f, 0.0f);指定使用(0, 0)纹理坐标。

通常，每个顶点使用不同的纹理，于是下面这样形式的代码是比较常见的。

```
glBegin( /* ... */ );
  glTexCoord2f( /* ... */ );  glVertex3f( /* ... */ );
  glTexCoord2f( /* ... */ );  glVertex3f( /* ... */ );
  /* ... */
glEnd();
```

> 当我们用一个坐标表示顶点在三维空间的位置时，可以使用 glRotate*等函数来对坐标进行转换。纹理坐标也可以进行这种转换。只要使用 glMatrixMode(GL_TEXTURE);，就可以切换到纹理矩阵（另外还有透视矩阵 GL_PROJECTION 和模型视图矩阵 GL_MODELVIEW，详细情况在第五课有讲述），然后 glRotate*，glScale*，glTranslate*等操作矩阵的函数就可以用来处理“对纹理坐标进行转换”的工作了。在简单应用中，可能不会对矩阵进行任何变换，这样考虑问题会比较简单。

3、纹理参数

到这里，入门所需要掌握的所有难点都被我们掌握了。但是，我们的知识仍然是不够的，如果仅利用现有的知识去使用纹理的话，你可能会发现纹理完全不起作用。这是因为在使用纹理前还有某些参数是必须设置的。

使用 glTexParameter*系列函数来设置纹理参数。通常需要设置下面四个参数：

GL_TEXTURE_MAG_FILTER：指当纹理图象被使用到一个大于它的形状上时（即：有可能纹理图象中的一个像素会被应用到实际绘制时的多个像素。例如将一幅 256*256 的纹理图象应用到一个 512*512 的正方形），应该如何处理。可选择的设置有 GL_NEAREST 和 GL_LINEAR，前者表示“使用纹理中坐标最接近的一个像素的颜色作为需要绘制的像素颜色”，后者表示“使用纹理中坐标最接近的若干个颜色，通过加权平均算法得到需要绘制的像素颜色”。前者只经过简单比较，需要运算较少，可能速度较快，后者需要经过加权平均计算，其中涉及除法运算，可能速度较慢（但如果有专门的处理硬件，也可能两者速度相同）。

从视觉效果上看，前者效果较差，在一些情况下锯齿现象明显，后者效果会较好（但如果纹理图象本身比较大，则两者在视觉效果上就会比较接近）。

GL_TEXTURE_MIN_FILTER：指当纹理图象被使用到一个小于（或等于）它的形状上时（即有可能纹理图象中的多个像素被应用到实际绘制时的一个像素。例如将一幅 256*256 的纹理图象应用到一个 128*128 的正方形），应该如何处理。可选择的设置有 GL_NEAREST，GL_LINEAR，GL_NEAREST_MIPMAP_NEAREST，GL_NEAREST_MIPMAP_LINEAR，GL_LINEAR_MIPMAP_NEAREST 和 GL_LINEAR_MIPMAP_LINEAR。其中后四个涉及到 mipmap，现在暂时不需要了解。前两个选项则和 GL_TEXTURE_MAG_FILTER 中的类似。此参数似乎是必须设置的（在我的计算机上，不设置此参数将得到错误的显示结果，但我目前并没有找到根据）。

GL_TEXTURE_WRAP_S：指当纹理坐标的第一维坐标值大于 1.0 或小于 0.0 时，应该如何处理。基本的选项有 GL_CLAMP 和 GL_REPEAT，前者表示“截断”，即超过 1.0 的按 1.0 处理，不足 0.0 的按 0.0 处理。后者表示“重复”，即对坐标值加上一个合适的整数（可以是正数或负数），得到一个在[0.0, 1.0]范围内的值，然后用这个值作为新的纹理坐标。例如：某二维纹理，在绘制某形状时，一像素需要得到纹理中坐标为(3.5, 0.5)的像素的颜色，其中第一维的坐标值 3.5 超过了 1.0，则在 GL_CLAMP 方式中将被转化为(1.0, 0.5)，在 GL_REPEAT 方式中将被转化为(0.5, 0.5)。在后来的 OpenGL 版本中，又增加了新的处理方式，这里不做介绍。如果不指定这个参数，则默认为 GL_REPEAT。

GL_TEXTURE_WRAP_T：指当纹理坐标的第二维坐标值大于 1.0 或小于 0.0 时，应该如何处理。选项与 GL_TEXTURE_WRAP_S 类似，不再重复。如果不指定这个参数，则默认为 GL_REPEAT。

设置参数的代码如下所示：

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
```

4、纹理对象

前面已经提到过，载入一幅纹理所需要的时间是比较多的。因此应该尽量减少载入纹理的次数。如果只有一幅纹理，则应该在第一次绘制前就载入它，以后就不需要再次载入了。这点与 glDrawPixels 函数很不相同。每次使用 glDrawPixels 函数，都需要把像素数据重新载入一次，因此用 glDrawPixels 函数来反复绘制图象的效率是较低的（如果只绘制一次，则不会有此问题），使用纹理来反复绘制图象是可取的做法。

但是，在每次绘制时要使用两幅或更多幅的纹理时，这个办法就行不通了。你可能会编写下面的代码：

```
glTexImage2D( /* ... */ ); // 载入第一幅纹理
// 使用第一幅纹理
glTexImage2D( /* ... */ ); // 载入第二幅纹理
```

// 使用第二幅纹理

// 当纹理的数量增加时，这段代码会变得更加复杂。

在绘制动画时，由于每秒钟需要将画面绘制数十次，因此如果使用上面的代码，就会反复载入纹理，这对计算机是非常大的负担，以目前的个人计算机配置来说，根本就无法让动画能够流畅的运行。因此，需要有一种机制，能够在不同的纹理之间进行快速的切换。

纹理对象正是这样一种机制。我们可以把每一幅纹理（包括纹理的像素数据、纹理大小等信息，也包括了前面所讲的纹理参数）放到一个纹理对象中，通过创建多个纹理对象来达到同时保存多幅纹理的目的。这样一来，在第一次使用纹理前，把所有的纹理都载入，然后在绘制时只需要指明究竟使用哪一个纹理对象就可以了。

使用纹理对象和使用显示列表有相似之处：使用一个正整数来作为纹理对象的编号。在使用前，可以调用 glGenTextures 来分配纹理对象。该函数有两种比较常见的用法：

GLuint texture_ID;

glGenTextures(1, &texture_ID); // 分配一个纹理对象的编号

或者：

GLuint texture_ID_list[5];

glGenTextures(5, texture_ID_list); // 分配 5 个纹理对象的编号

零是一个特殊的纹理对象编号，表示“默认的纹理对象”，在分配正确的情况下，glGenTextures 不会分配这个编号。与 glGenTextures 对应的是 glDeleteTextures，用于销毁一个纹理对象。

在分配了纹理对象编号后，使用 glBindTexture 函数来指定“当前所使用的纹理对象”。然后就可以使用 glTexImage*系列函数来指定纹理像素、使用 glTexParameter*系列函数来指定纹理参数、使用 glTexCoord*系列函数来指定纹理坐标了。如果不使用 glBindTexture 函数，那么 glTexImage*、glTexParameter*、glTexCoord*系列函数默认在一个编号为 0 的纹理对象上进行操作。glBindTexture 函数有两个参数，第一个参数是需

要使用纹理的目标，因为我们现在只学习二维纹理，所以指定为 GL_TEXTURE_2D，第二个参数是所使用的纹理的编号。

使用多个纹理对象，就可以使 OpenGL 同时保存多个纹理。在使用时只需要调用 glBindTexture 函数，在不同纹理之间进行切换，而不需要反复载入纹理，因此动画的绘制速度会有非常明显的提升。典型的代码如下所示：

```
// 在程序开始时：分配好纹理编号，并载入纹理
glGenTextures( /* ... */ );
glBindTexture(GL_TEXTURE_2D, texture_ID_1);
// 载入第一幅纹理
glBindTexture(GL_TEXTURE_2D, texture_ID_2);
// 载入第二幅纹理


// 在绘制时，切换并使用纹理，不需要再进行载入
glBindTexture(GL_TEXTURE_2D, texture_ID_1); // 指定第一幅纹理
// 使用第一幅纹理
glBindTexture(GL_TEXTURE_2D, texture_ID_2); // 指定第二幅纹理
// 使用第二幅纹理
```

> 提示：纹理对象是从 OpenGL 1.1 版开始才有的，最旧版本的 OpenGL 1.0 并没有处理纹理对象的功能。不过，我想各位的机器不会是比 OpenGL 1.1 更低的版本（Windows 95 就自带了 OpenGL 1.1 版本，遗憾的是，Microsoft 对 OpenGL 的支持并不积极，Windows XP 也还采用 1.1 版本。据说 Vista 使用的是 OpenGL 1.4 版。当然了，如果安装显卡驱动的话，现在的主流显卡一般都附带了适用于该显卡的 OpenGL 1.4 版或更高版本），所以这个问题也就不算是问题了。

5、示例程序

纹理入门所需要掌握的知识点就介绍到这里了。但是如果不实际动手操作的话，也是不可能真正掌握的。下面我们来看看本课开头的那个纹理效果是如何实现的吧。

因为代码比较长，我把它拆分成了三段，大家如果要编译的话，应该把三段代码按顺序连在一起编译。如果要运行的话，除了要保证有一个名称为 dummy.bmp，图象大小为 1*1 的 24 位 BMP 文件，还要把本课开始的两幅纹理图片保存到正确位置（一幅名叫 ground.bmp，另一幅名叫 wall.bmp。注意：我为了节省网络空间，把两幅图片都转成 jpg 格式了，读者把图片保存到本地后，需要把它们再转化为 BMP 格式。可以使用 Windows XP 带的画图程序中的“另存为”功能完成这一转换）。

第一段代码如下。其中的主体——grab 函数，是我们在第十课介绍过的，这里仅仅是抄过来用一下，目的是为了将最终效果图保存到一个名字叫 grab.bmp 的文件中。（当然了，为了保证程序的正确运行，那个大小为 1*1 的 dummy.bmp 文件仍然是必要的，参见第十课）

```
#define WindowWidth  400
#define WindowHeight 400
#define WindowTitle  "OpenGL 纹理测试"

#include <gl/glut.h>
#include <stdio.h>
#include <stdlib.h>

/* 函数 grab
 * 抓取窗口中的像素
 * 假设窗口宽度为 WindowWidth，高度为 WindowHeight
 */
#define BMP_Header_Length 54
void grab(void)
{
   FILE*    pDummyFile;
   FILE*    pWritingFile;
   GLubyte* pPixelData;
   GLubyte  BMP_Header[BMP_Header_Length];
   GLint    i, j;
   GLint    PixelDataLength;

   // 计算像素数据的实际长度
   i = WindowWidth * 3;  // 得到每一行的像素数据长度
   while( i%4 != 0 )     // 补充数据，直到 i 是的倍数
      ++i;               // 本来还有更快的算法，
                         // 但这里仅追求直观，对速度没有太高要求
```

```
PixelDataLength = i * WindowHeight;

// 分配内存和打开文件
pPixelData = (GLubyte*)malloc(PixelDataLength);
if( pPixelData == 0 )
    exit(0);

pDummyFile = fopen("dummy.bmp", "rb");
if( pDummyFile == 0 )
    exit(0);

pWritingFile = fopen("grab.bmp", "wb");
if( pWritingFile == 0 )
    exit(0);

// 读取像素
glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
glReadPixels(0, 0, WindowWidth, WindowHeight,
    GL_BGR_EXT, GL_UNSIGNED_BYTE, pPixelData);

// 把 dummy.bmp 的文件头复制为新文件的文件头
fread(BMP_Header, sizeof(BMP_Header), 1, pDummyFile);
fwrite(BMP_Header, sizeof(BMP_Header), 1, pWritingFile);
fseek(pWritingFile, 0x0012, SEEK_SET);
i = WindowWidth;
j = WindowHeight;
fwrite(&i, sizeof(i), 1, pWritingFile);
fwrite(&j, sizeof(j), 1, pWritingFile);

// 写入像素数据
```

```
    fseek(pWritingFile, 0, SEEK_END);
    fwrite(pPixelData, PixelDataLength, 1, pWritingFile);

    // 释放内存和关闭文件
    fclose(pDummyFile);
    fclose(pWritingFile);
    free(pPixelData);
}
```

第二段代码是我们的重点。它包括两个函数。其中 power_of_two 比较简单，虽然实现手段有点奇特，但也并非无法理解（即使真的无法理解，读者也可以给出自己的解决方案，用一些循环以及多使用一些位操作也没关系。反正，这里不是重点啦）。另一个 load_texture 函数却是重头戏：打开 BMP 文件、读取其中的高度和宽度信息、计算像素数据所占的字节数、为像素数据分配空间、读取像素数据、对像素图象进行缩放（如果必要的话）、分配新的纹理编号、填写纹理参数、载入纹理，所有的功能都在同一个函数里面完成了。为了叙述方便，我把所有的解释都放在了注释里。

```
/* 函数 power_of_two
 * 检查一个整数是否为 2 的整数次方，如果是，返回 1，否则返回 0
 * 实际上只要查看其二进制位中有多少个，如果正好有 1 个，返回 1，否则返回 0
 * 在“查看其二进制位中有多少个”时使用了一个小技巧
 * 使用 n &= (n-1)可以使得 n 中的减少一个（具体原理大家可以自己思考）
 */
int power_of_two(int n)
{
    if( n <= 0 )
        return 0;
    return (n & (n-1)) == 0;
}

/* 函数 load_texture
 * 读取一个 BMP 文件作为纹理
 * 如果失败，返回 0，如果成功，返回纹理编号
```

```
*/
GLuint load_texture(const char* file_name)
{
    GLint width, height, total_bytes;
    GLubyte* pixels = 0;
    GLuint last_texture_ID, texture_ID = 0;

    // 打开文件，如果失败，返回
    FILE* pFile = fopen(file_name, "rb");
    if( pFile == 0 )
        return 0;

    // 读取文件中图象的宽度和高度
    fseek(pFile, 0x0012, SEEK_SET);
    fread(&width, 4, 1, pFile);
    fread(&height, 4, 1, pFile);
    fseek(pFile, BMP_Header_Length, SEEK_SET);

    // 计算每行像素所占字节数，并根据此数据计算总像素字节数
    {
        GLint line_bytes = width * 3;
        while( line_bytes % 4 != 0 )
            ++line_bytes;
        total_bytes = line_bytes * height;
    }

    // 根据总像素字节数分配内存
    pixels = (GLubyte*)malloc(total_bytes);
    if( pixels == 0 )
    {
```

```
    fclose(pFile);
    return 0;
}

// 读取像素数据
if( fread(pixels, total_bytes, 1, pFile) <= 0 )
{
    free(pixels);
    fclose(pFile);
    return 0;
}

// 在旧版本的 OpenGL 中
// 如果图象的宽度和高度不是的整数次方，则需要进行缩放
// 这里并没有检查 OpenGL 版本，出于对版本兼容性的考虑，按旧版本处理
// 另外，无论是旧版本还是新版本，
// 当图象的宽度和高度超过当前 OpenGL 实现所支持的最大值时，也要进行缩放
{
    GLint max;
    glGetIntegerv(GL_MAX_TEXTURE_SIZE, &max);
    if( !power_of_two(width)
     || !power_of_two(height)
     || width > max
     || height > max )
    {
        const GLint new_width = 256;
        const GLint new_height = 256; // 规定缩放后新的大小为边长的正方形
        GLint new_line_bytes, new_total_bytes;
        GLubyte* new_pixels = 0;
```

```
        // 计算每行需要的字节数和总字节数
        new_line_bytes = new_width * 3;
        while( new_line_bytes % 4 != 0 )
            ++new_line_bytes;
        new_total_bytes = new_line_bytes * new_height;

        // 分配内存
        new_pixels = (GLubyte*)malloc(new_total_bytes);
        if( new_pixels == 0 )
        {
            free(pixels);
            fclose(pFile);
            return 0;
        }

        // 进行像素缩放
        gluScaleImage(GL_RGB,
            width, height, GL_UNSIGNED_BYTE, pixels,
            new_width, new_height, GL_UNSIGNED_BYTE, new_pixels);

        // 释放原来的像素数据，把 pixels 指向新的像素数据，并重新设置 width 和 height
        free(pixels);
        pixels = new_pixels;
        width = new_width;
        height = new_height;
    }
}

// 分配一个新的纹理编号
glGenTextures(1, &texture_ID);
```

```
    if( texture_ID == 0 )
    {
        free(pixels);
        fclose(pFile);
        return 0;
    }

    // 绑定新的纹理，载入纹理并设置纹理参数
    // 在绑定前，先获得原来绑定的纹理编号，以便在最后进行恢复
    glGetIntegerv(GL_TEXTURE_BINDING_2D, &last_texture_ID);
    glBindTexture(GL_TEXTURE_2D, texture_ID);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
    glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0,
        GL_BGR_EXT, GL_UNSIGNED_BYTE, pixels);
    glBindTexture(GL_TEXTURE_2D, last_texture_ID);

    // 之前为 pixels 分配的内存可在使用 glTexImage2D 以后释放
    // 因为此时像素数据已经被 OpenGL 另行保存了一份（可能被保存到专门的图形硬件中）
    free(pixels);
    return texture_ID;
}
```

第三段代码是关于显示的部分，以及 main 函数。注意，我们只在 main 函数中读取了两幅纹理，并把它们保存在各自的纹理对象中，以后就再也不载入纹理。每次绘制时使用 glBindTexture 在不同的纹理对象中切换。另外，我们使用了超过 1.0 的纹理坐标，由于 GL_TEXTURE_WRAP_S 和 GL_TEXTURE_WRAP_T 参数都被设置为 GL_REPEAT，所以得到的效果就是纹理像素的重复，有点向地板砖的花纹那样。读者可以试着修改“墙”的纹理坐标，将 5.0 修改为 10.0，看看效果有什么变化。

```
/* 两个纹理对象的编号
 */
GLuint texGround;
GLuint texWall;

void display(void)
{
   // 清除屏幕
   glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

   // 设置视角
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   gluPerspective(75, 1, 1, 21);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
   gluLookAt(1, 5, 5, 0, 0, 0, 0, 0, 1);

   // 使用“地”纹理绘制土地
   glBindTexture(GL_TEXTURE_2D, texGround);
   glBegin(GL_QUADS);
      glTexCoord2f(0.0f, 0.0f); glVertex3f(-8.0f, -8.0f, 0.0f);
      glTexCoord2f(0.0f, 5.0f); glVertex3f(-8.0f, 8.0f, 0.0f);
      glTexCoord2f(5.0f, 5.0f); glVertex3f(8.0f, 8.0f, 0.0f);
      glTexCoord2f(5.0f, 0.0f); glVertex3f(8.0f, -8.0f, 0.0f);
   glEnd();
   // 使用“墙”纹理绘制栅栏
   glBindTexture(GL_TEXTURE_2D, texWall);
   glBegin(GL_QUADS);
      glTexCoord2f(0.0f, 0.0f); glVertex3f(-6.0f, -3.0f, 0.0f);
```

```
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-6.0f, -3.0f, 1.5f);
        glTexCoord2f(5.0f, 1.0f); glVertex3f(6.0f, -3.0f, 1.5f);
        glTexCoord2f(5.0f, 0.0f); glVertex3f(6.0f, -3.0f, 0.0f);
    glEnd();

    // 旋转后再绘制一个
    glRotatef(-90, 0, 0, 1);
    glBegin(GL_QUADS);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-6.0f, -3.0f, 0.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-6.0f, -3.0f, 1.5f);
        glTexCoord2f(5.0f, 1.0f); glVertex3f(6.0f, -3.0f, 1.5f);
        glTexCoord2f(5.0f, 0.0f); glVertex3f(6.0f, -3.0f, 0.0f);
    glEnd();

    // 交换缓冲区，并保存像素数据到文件
    glutSwapBuffers();
    grab();
}

int main(int argc, char* argv[])
{
    // GLUT 初始化
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100, 100);
    glutInitWindowSize(WindowWidth, WindowHeight);
    glutCreateWindow(WindowTitle);
    glutDisplayFunc(&display);

    // 在这里做一些初始化
```

```
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_TEXTURE_2D);
    texGround = load_texture("ground.bmp");
    texWall = load_texture("wall.bmp");

    // 开始显示
    glutMainLoop();

    return 0;
}
```

小结：

本课介绍了 OpenGL 纹理的入门知识。

利用纹理可以进行比 glReadPixels 和 glDrawPixels 更复杂的像素绘制，因此可以实现很多精彩的效果。

本课只涉及了二维纹理。OpenGL 还支持一维和三维纹理，其原理是类似的。

在使用纹理前，要启用纹理。并且，还需要将像素数据载入到纹理中。注意纹理的宽度和高度，目前很多 OpenGL 的实现都还要求其值为 2 的整数次方，如果纹理图象本身并不满足这个条件，可以使用 gluScaleImage 函数来进行缩放。为了正确的使用纹理，需要设置纹理参数。

载入纹理所需要的系统开销是比较大的，应该尽可能减少载入纹理的次数。如果程序中只使用一幅纹理，则只在第一次使用前载入，以后不必重新载入。如果程序中要使用多幅纹理，不应该反复载入它们，而应该将每个纹理都用一个纹理对象来保存，并使用 glBindTextures 在各个纹理之间进行切换。

本课还给出了一个程序（到目前为止，它是这个 OpenGL 教程系列中所给出的程序中最长的）。该程序演示了纹理的基本使用方法，本课程涉及到的几乎所有内容都被包括其中，这是对本课中文字说明的一个补充。如果读者有什么不明白的地方，也可以以这个程序作为参考。

12.第十二课：

片断测试其实就是测试每一个像素，只有通过测试的像素才会被绘制，没有通过测试的像素则不进行绘制。OpenGL 提供了多种测试操作，利用这些操作可以实现一些特殊的效果。

我们在前面的课程中，曾经提到了“深度测试”的概念，它在绘制三维场景的时候特别有用。在不使用深度测试的时候，如果我们先绘制一个距离较近的物体，再绘制距离较远的物体，则距离远的物体因为后绘制，会把距离近的物体覆盖掉，这样的效果并不是我们所希望的。

如果使用了深度测试，则情况就会有所不同：每当一个像素被绘制，OpenGL 就记录这个像素的“深度”（深度可以理解为：该像素距离观察者的距离。深度值越大，表示距离越远），如果有新的像素即将覆盖原来的像素时，深度测试会检查新的深度是否会比原来的深度值小。如果是，则覆盖像素，绘制成功；如果不是，

则不会覆盖原来的像素，绘制被取消。这样一来，即使我们先绘制比较近的物体，再绘制比较远的物体，则远的物体也不会覆盖近的物体了。

实际上，只要存在深度缓冲区，无论是否启用深度测试，OpenGL 在像素被绘制时都会尝试将深度数据写入到缓冲区内，除非调用了 glDepthMask(GL_FALSE)来禁止写入。这些深度数据除了用于常规的测试外，还可以有一些有趣的用途，比如绘制阴影等等。

除了深度测试，OpenGL 还提供了剪裁测试、Alpha 测试和模板测试。

1、剪裁测试

剪裁测试用于限制绘制区域。我们可以指定一个矩形的剪裁窗口，当启用剪裁测试后，只有在这个窗口之内的像素才能被绘制，其它像素则会被丢弃。换句话说，无论怎么绘制，剪裁窗口以外的像素将不会被修改。

有的朋友可能玩过《魔兽争霸 3》这款游戏。游戏时如果选中一个士兵，则画面下方的一个方框内就会出现该士兵的头像。为了保证该头像无论如何绘制都不会越界而覆盖到外面的像素，就可以使用剪裁测试。

可以通过下面的代码来启用或禁用剪裁测试：

```
glEnable(GL_SCISSOR_TEST); // 启用剪裁测试
glDisable(GL_SCISSOR_TEST); // 禁用剪裁测试
```

可以通过下面的代码来指定一个位置在(x, y)，宽度为 width，高度为 height 的剪裁窗口。

```
glScissor(x, y, width, height);
```

> 注意，OpenGL 窗口坐标是以左下角为(0, 0)，右上角为(width, height)的，这与 Windows 系统窗口有所不同。
>
> 还有一种方法可以保证像素只绘制到某一个特定的矩形区域内，这就是视口变换（在第五课第 3 节中有介绍）。但视口变换和剪裁测试是不同的。视口变换是将所有内容缩放到合适的大小后，放到一个矩形的区域内；而剪裁测试不会进行缩放，超出矩形范围的像素直接忽略掉。

2、Alpha 测试

在前面的课程中，我们知道像素的 Alpha 值可以用于混合操作。其实 Alpha 值还有一个用途，这就是 Alpha 测试。当每个像素即将绘制时，如果启动了 Alpha 测试，OpenGL 会检查像素的 Alpha 值，只有 Alpha 值满足条件的像素才会进行绘制（严格的说，满足条件的像素会通过本项测试，进行下一种测试，只有所有

测试都通过，才能进行绘制），不满足条件的则不进行绘制。这个“条件”可以是：始终通过（默认情况）、始终不通过、大于设定值则通过、小于设定值则通过、等于设定值则通过、大于等于设定值则通过、小于等于设定值则通过、不等于设定值则通过。

如果我们需要绘制一幅图片，而这幅图片的某些部分又是透明的（想象一下，你先绘制一幅相片，然后绘制一个相框，则相框这幅图片有很多地方都是透明的，这样就可以透过相框看到下面的照片），这时可以使用 Alpha 测试。将图片中所有需要透明的地方的 Alpha 值设置为 0.0，不需要透明的地方 Alpha 值设置为 1.0，然后设置 Alpha 测试的通过条件为：“大于 0.5 则通过”，这样便能达到目的。当然也可以设置需要透明的地方 Alpha 值为 1.0，不需要透明的地方 Alpha 值设置为 0.0，然后设置条件为“小于 0.5 则通过”。Alpha 测试的设置方式往往不只一种，可以根据个人喜好和实际情况需要进行选择。

可以通过下面的代码来启用或禁用 Alpha 测试：

```
glEnable(GL_ALPHA_TEST);  // 启用 Alpha 测试
glDisable(GL_ALPHA_TEST); // 禁用 Alpha 测试
```

可以通过下面的代码来设置 Alpha 测试条件为“大于 0.5 则通过”：

```
glAlphaFunc(GL_GREATER, 0.5f);
```

> 该函数的第二个参数表示设定值，用于进行比较。第一个参数是比较方式，除了 GL_LESS(小于则通过)外，还可以选择：
> GL_ALWAYS（始终通过），
> GL_NEVER（始终不通过），
> GL_LESS（小于则通过），
> GL_LEQUAL（小于等于则通过），
> GL_EQUAL（等于则通过），
> GL_GEQUAL（大于等于则通过），
> GL_NOTEQUAL（不等于则通过）。

现在我们来看一个实际例子。一幅照片图片，一幅相框图片，如何将它们组合在一起呢？为了简单起见，我们使用前面两课一直使用的 24 位 BMP 文件来作为图片格式。（因为发布到网络上，为了节约容量，我所发布的是 JPG 格式。大家下载后可以用 Windows XP 自带的画图工具打开，并另存为 24 位 BMP 格式）

注：第一幅图片是著名网络游戏《魔兽世界》的一幅桌面背景，用在这里希望没有涉及版权问题。如果有什么不妥，请及时指出，我会立即更换。

在 24 位的 BMP 文件格式中，BGR 三种颜色各占 8 位，没有保存 Alpha 值，因此无法直接使用 Alpha 测试。注意到相框那幅图片中，所有需要透明的位置都是白色，所以我们在程序中设置所有白色（或很接近白色）的像素 Alpha 值为 0.0，设置其它像素 Alpha 值为 1.0，然后设置 Alpha 测试的条件为“大于 0.5 则通过”即可。这种使用某种特殊颜色来代表透明颜色的技术，有时又被成为 Color Key 技术。

利用前面第 11 课的一段代码，将图片读取为纹理，然后利用下面这个函数来设置“当前纹理”中每一个像素的 Alpha 值。

```
/* 将当前纹理 BGR 格式转换为 BGRA 格式
 * 纹理中像素的 RGB 值如果与指定 rgb 相差不超过 absolute，则将 Alpha 设置为 0.0，否则设置为 1.0
 */
void texture_colorkey(GLubyte r, GLubyte g, GLubyte b, GLubyte absolute)
{
    GLint width, height;
    GLubyte* pixels = 0;

    // 获得纹理的大小信息
    glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_WIDTH, &width);
    glGetTexLevelParameteriv(GL_TEXTURE_2D, 0, GL_TEXTURE_HEIGHT, &height);

    // 分配空间并获得纹理像素
```

```
    pixels = (GLubyte*)malloc(width*height*4);
    if( pixels == 0 )
        return;
    glGetTexImage(GL_TEXTURE_2D, 0, GL_BGRA_EXT, GL_UNSIGNED_BYTE, pixels);

    // 修改像素中的 Alpha 值
    // 其中 pixels[i*4], pixels[i*4+1], pixels[i*4+2], pixels[i*4+3]
    //  分别表示第 i 个像素的蓝、绿、红、Alpha 四种分量，0 表示最小，255 表示最大
    {
        GLint i;
        GLint count = width * height;
        for(i=0; i<count; ++i)
        {
            if( abs(pixels[i*4] - b) <= absolute
             && abs(pixels[i*4+1] - g) <= absolute
             && abs(pixels[i*4+2] - r) <= absolute )
                pixels[i*4+3] = 0;
            else
                pixels[i*4+3] = 255;
        }
    }

    // 将修改后的像素重新设置到纹理中，释放内存
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width, height, 0,
        GL_BGRA_EXT, GL_UNSIGNED_BYTE, pixels);
    free(pixels);
}
```

有了纹理后，我们开启纹理，指定合适的纹理坐标并绘制一个矩形，这样就可以在屏幕上将图片绘制出来。

我们先绘制相片的纹理，再绘制相框的纹理。程序代码如下：

```
void display(void)
{
    static int initialized  = 0;
    static GLuint texWindow  = 0;
    static GLuint texPicture = 0;

    // 执行初始化操作，包括：读取相片，读取相框，将相框由 BGR 颜色转换为 BGRA，启用二维纹理
    if( !initialized )
    {
        texPicture = load_texture("pic.bmp");
        texWindow  = load_texture("window.bmp");
        glBindTexture(GL_TEXTURE_2D, texWindow);
        texture_colorkey(255, 255, 255, 10);

        glEnable(GL_TEXTURE_2D);

        initialized = 1;
    }

    // 清除屏幕
    glClear(GL_COLOR_BUFFER_BIT);

    // 绘制相片，此时不需要进行 Alpha 测试，所有的像素都进行绘制
    glBindTexture(GL_TEXTURE_2D, texPicture);
    glDisable(GL_ALPHA_TEST);
    glBegin(GL_QUADS);
        glTexCoord2f(0, 0);     glVertex2f(-1.0f, -1.0f);
        glTexCoord2f(0, 1);     glVertex2f(-1.0f,  1.0f);
        glTexCoord2f(1, 1);     glVertex2f( 1.0f,  1.0f);
        glTexCoord2f(1, 0);     glVertex2f( 1.0f, -1.0f);
```

```
    glEnd();

    // 绘制相框，此时进行 Alpha 测试，只绘制不透明部分的像素
    glBindTexture(GL_TEXTURE_2D, texWindow);
    glEnable(GL_ALPHA_TEST);
    glAlphaFunc(GL_GREATER, 0.5f);
    glBegin(GL_QUADS);
        glTexCoord2f(0, 0);     glVertex2f(-1.0f, -1.0f);
        glTexCoord2f(0, 1);     glVertex2f(-1.0f,  1.0f);
        glTexCoord2f(1, 1);     glVertex2f( 1.0f,  1.0f);
        glTexCoord2f(1, 0);     glVertex2f( 1.0f, -1.0f);
    glEnd();

    // 交换缓冲
    glutSwapBuffers();
}
```

其中：load_texture 函数是从第 11 课中照搬过来的（该函数还使用了一个 power_of_two 函数，一个 BMP_Header_Length 常数，同样照搬），无需进行修改。main 函数跟其它课程的基本相同，不再重复。

程序运行后，会发现相框与相片的衔接有些不自然，这是因为相框某些边缘部分虽然肉眼看上去是白色，但其实 RGB 值与纯白色相差并不少，因此程序计算其 Alpha 值时认为其不需要透明。解决办法是仔细处理相框中的每个像素，在需要透明的地方涂上纯白色，这也许是一件很需要耐心的工作。

大家可能会想：前面我们学习过混合操作，混合可以实现半透明，自然也可以通过设定实现全透明。也就是说，Alpha 测试可以实现的效果几乎都可以通过 OpenGL 混合功能来实现。那么为什么还需要一个 Alpha 测试呢？答案就是，这与性能相关。Alpha 测试只要简单的比较大小就可以得到最终结果，而混合操作一般需要进行乘法运算，性能有所下降。另外，OpenGL 测试的顺序是：剪裁测试、Alpha 测试、模板测试、深度测试。如果某项测试不通过，则不会进行下一步，而只有所有测试都通过的情况下才会执行混合操作。因此，在使用 Alpha 测试的情况下，透明的像素就不需要经过模板测试和深度测试了；而如果使用混合操作，即使透明的像素也需要进行模板测试和深度测试，性能会有所下降。还有一点：对于那些“透明”的像素来说，如果使用 Alpha 测试，则“透明”的像素不会通过测试，因此像素的深度值不会被修改；而使用混合操作时，虽然像素的颜色没有被修改，但它的深度值则有可能被修改掉了。

因此，如果所有的像素都是“透明”或“不透明”，没有“半透明”时，应该尽量采用 Alpha 测试而不是采用混合操作。当需要绘制半透明像素时，才采用混合操作。

3、模板测试

模板测试是所有 OpenGL 测试中比较复杂的一种。

首先，模板测试需要一个模板缓冲区，这个缓冲区是在初始化 OpenGL 时指定的。如果使用 GLUT 工具包，可以在调用 glutInitDisplayMode 函数时在参数中加上 GLUT_STENCIL，例如：

```
glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGBA | GLUT_STENCIL);
```

在 Windows 操作系统中，即使没有明确要求使用模板缓冲区，有时候也会分配模板缓冲区。但为了保证程序的通用性，最好还是明确指定使用模板缓冲区。如果确实没有分配模板缓冲区，则所有进行模板测试的像素全部都会通过测试。

通过 glEnable/glDisable 可以启用或禁用模板测试。

```
glEnable(GL_STENCIL_TEST);  // 启用模板测试
glDisable(GL_STENCIL_TEST); // 禁用模板测试
```

OpenGL 在模板缓冲区中为每个像素保存了一个“模板值”，当像素需要进行模板测试时，将设定的模板参考值与该像素的“模板值”进行比较，符合条件的通过测试，不符合条件的则被丢弃，不进行绘制。
条件的设置与 Alpha 测试中的条件设置相似。但注意 Alpha 测试中是用浮点数来进行比较，而模板测试则是用整数来进行比较。比较也有八种情况：始终通过、始终不通过、大于则通过、小于则通过、大于等于则通过、小于等于则通过、等于则通过、不等于则通过。

```
glStencilFunc(GL_LESS, 3, mask);
```

这段代码设置模板测试的条件为：“小于 3 则通过”。glStencilFunc 的前两个参数意义与 glAlphaFunc 的两个参数类似，第三个参数的意义为：如果进行比较，则只比较 mask 中二进制为 1 的位。例如，某个像素模板值为 5（二进制 101），而 mask 的二进制值为 00000011，因为只比较最后两位，5 的最后两位为 01，其实是小于 3 的，因此会通过测试。

如何设置像素的“模板值”呢？glClear 函数可以将所有像素的模板值复位。代码如下：

```
glClear(GL_STENCIL_BUFFER_BIT);
```

可以同时复位颜色值和模板值：

glClear(GL_COLOR_BUFFER_BIT | GL_STENCIL_BUFFER_BIT);

正如可以使用 glClearColor 函数来指定清空屏幕后的颜色那样，也可以使用 glClearStencil 函数来指定复位后的“模板值”。

每个像素的“模板值”会根据模板测试的结果和深度测试的结果而进行改变。

glStencilOp(fail, zfail, zpass);

该函数指定了三种情况下“模板值”该如何变化。第一个参数表示模板测试未通过时该如何变化；第二个参数表示模板测试通过，但深度测试未通过时该如何变化；第三个参数表示模板测试和深度测试均通过时该如何变化。如果没有起用模板测试，则认为模板测试总是通过；如果没有启用深度测试，则认为深度测试总是通过）
变化可以是：
GL_KEEP（不改变，这也是默认值），
GL_ZERO（回零），
GL_REPLACE（使用测试条件中的设定值来代替当前模板值），
GL_INCR（增加 1，但如果已经是最大值，则保持不变），
GL_INCR_WRAP（增加 1，但如果已经是最大值，则从零重新开始），
GL_DECR（减少 1，但如果已经是零，则保持不变），
GL_DECR_WRAP（减少 1，但如果已经是零，则重新设置为最大值），
GL_INVERT（按位取反）。

在新版本的 OpenGL 中，允许为多边形的正面和背面使用不同的模板测试条件和模板值改变方式，于是就有了 glStencilFuncSeparate 函数和 glStencilOpSeparate 函数。这两个函数分别与 glStencilFunc 和 glStencilOp 类似，只在最前面多了一个参数 face，用于指定当前设置的是哪个面。可以选择 GL_FRONT, GL_BACK, GL_FRONT_AND_BACK。

注意：模板缓冲区与深度缓冲区有一点不同。无论是否启用深度测试，当有像素被绘制时，总会重新设置该像素的深度值（除非设置 glDepthMask(GL_FALSE);）。而模板测试如果不启用，则像素的模板值会保持不变，只有启用模板测试时才有可能修改像素的模板值。（这一结论是我自己的实验得出的，暂时没发现什么资料上是这样写。如果有不正确的地方，欢迎指正）
另外，模板测试虽然是从 OpenGL 1.0 就开始提供的功能，但是对于个人计算机而言，硬件实现模板测试的似乎并不多，很多计算机系统直接使用 CPU 运算来完成模板测试。因此在一些老的显卡，或者是多数集成显卡上，大量而频繁的使用模板测试可能造成程序运行效率低下。即使是当前配置比较高端的个人计算机，也尽量不要使用 glStencilFuncSeparate 和 glStencilOpSeparate 函数。

从前面所讲可以知道，使用剪裁测试可以把绘制区域限制在一个矩形的区域内。但如果需要把绘制区域限制在一个不规则的区域内，则需要使用模板测试。

例如：绘制一个湖泊，以及周围的树木，然后绘制树木在湖泊中的倒影。为了保证倒影被正确的限制在湖泊表面，可以使用模板测试。具体的步骤如下：

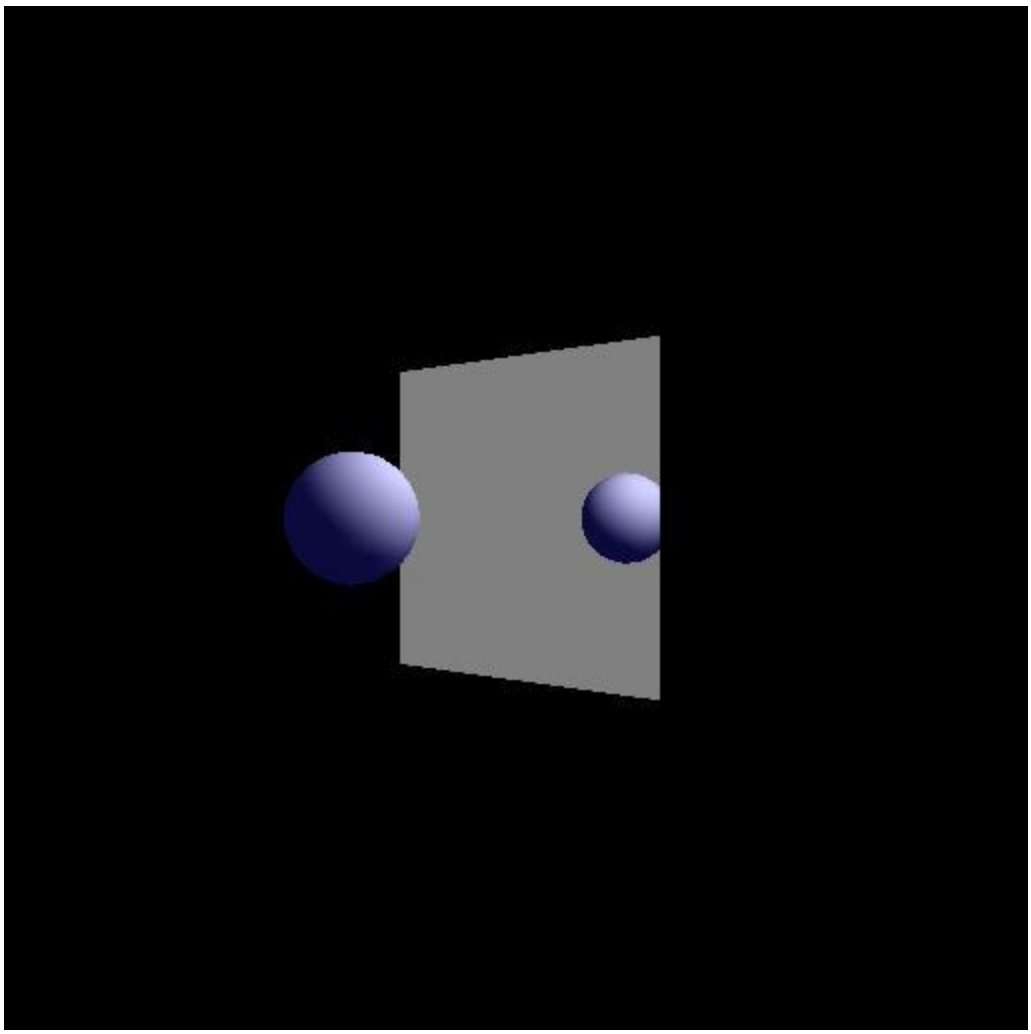
(1) 关闭模板测试，绘制地面和树木。

(2) 开启模板测试，使用 glClear 设置所有像素的模板值为 0。

(3) 设置 glStencilFunc(GL_ALWAYS, 1, 1); glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);绘制湖泊水面。这样一来，湖泊水面的像素的“模板值”为 1，而其它地方像素的“模板值”为 0。

(4) 设置 glStencilFunc(GL_EQUAL, 1, 1); glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);绘制倒影。这样一来，只有“模板值”为 1 的像素才会被绘制，因此只有“水面”的像素才有可能被倒影的像素替换，而其它像素则保持不变。

我们仍然来看一个实际的例子。这是一个比较简单的场景：空间中有一个球体，一个平面镜。我们站在某个特殊的观察点，可以看到球体在平面镜中的镜像，并且镜像处于平面镜的边缘，有一部分因为平面镜大小的限制，而无法显示出来。整个场景的效果如下图：



绘制这个场景的思路跟前面提到的湖面倒影是接近的。

假设平面镜所在的平面正好是 X 轴和 Y 轴所确定的平面，则球体和它在平面镜中的镜像是关于这个平面对称的。我们用一个 draw_sphere 函数来绘制球体，先调用该函数以绘制球体本身，然后调用 glScalef(1.0f, 1.

0f, -1.0f); 再调用 draw_sphere 函数，就可以绘制球体的镜像。

另外需要注意的地方就是：因为是绘制三维的场景，我们开启了深度测试。但是站在观察者的位置，球体的镜像其实是在平面镜的“背后”，也就是说，如果按照常规的方式绘制，平面镜会把镜像覆盖掉，这不是我们想要的效果。解决办法就是：设置深度缓冲区为只读，绘制平面镜，然后设置深度缓冲区为可写的状态，绘制平面镜“背后”的镜像。

有的朋友可能会问：如果在绘制镜像的时候关闭深度测试，那镜像不就不会被平面镜遮挡了吗？为什么还要开启深度测试，又需要把深度缓冲区设置为只读呢？实际情况是：虽然关闭深度测试确实可以让镜像不被平面镜遮挡，但是镜像本身会出现若干问题。我们看到的镜像是一个球体，但实际上这个球体是由很多的多边形所组成的，这些多边形有的代表了我们所能看到的“正面”，有的则代表了我们不能看到的“背面”。如果关闭深度测试，而有的“背面”多边形又比“正面”多边形先绘制，就会造成球体的背面反而把正面挡住了，这不是我们想要的效果。为了确保正面可以挡住背面，应该开启深度测试。

绘制部分的代码如下：

```
void draw_sphere()
{
  // 设置光源
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  {
    GLfloat
      pos[]    = {5.0f, 5.0f, 0.0f, 1.0f},
      ambient[] = {0.0f, 0.0f, 1.0f, 1.0f};
    glLightfv(GL_LIGHT0, GL_POSITION, pos);
    glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
  }

  // 绘制一个球体
  glColor3f(1, 0, 0);
  glPushMatrix();
  glTranslatef(0, 0, 2);
  glutSolidSphere(0.5, 20, 20);
```

```
    glPopMatrix();
}

void display(void)
{
    // 清除屏幕
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // 设置观察点
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60, 1, 5, 25);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(5, 0, 6.5, 0, 0, 0, 0, 1, 0);

    glEnable(GL_DEPTH_TEST);

    // 绘制球体
    glDisable(GL_STENCIL_TEST);
    draw_sphere();

    // 绘制一个平面镜。在绘制的同时注意设置模板缓冲。
    // 另外，为了保证平面镜之后的镜像能够正确绘制，在绘制平面镜时需要将深度缓冲区设置为只读的。
    // 在绘制时暂时关闭光照效果
    glClearStencil(0);
    glClear(GL_STENCIL_BUFFER_BIT);
    glStencilFunc(GL_ALWAYS, 1, 0xFF);
    glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
    glEnable(GL_STENCIL_TEST);
```

```
    glDisable(GL_LIGHTING);
    glColor3f(0.5f, 0.5f, 0.5f);
    glDepthMask(GL_FALSE);
    glRectf(-1.5f, -1.5f, 1.5f, 1.5f);
    glDepthMask(GL_TRUE);

    // 绘制一个与先前球体关于平面镜对称的球体，注意光源的位置也要发生对称改变
    // 因为平面镜是在 X 轴和 Y 轴所确定的平面，所以只要 Z 坐标取反即可实现对称
    // 为了保证球体的绘制范围被限制在平面镜内部，使用模板测试
    glStencilFunc(GL_EQUAL, 1, 0xFF);
    glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);
    glScalef(1.0f, 1.0f, -1.0f);
    draw_sphere();

    // 交换缓冲
    glutSwapBuffers();

    // 截图
    grab();
}
```

其中 display 函数的末尾调用了一个 grab 函数，它保存当前的图象到一个 BMP 文件。这个函数本来是在第十课和第十一课中都有所使用的。但是我发现它有一个 bug，现在进行了修改：在函数最开头的部分加上一句：glReadBuffer(GL_FRONT);即可。注意这个函数最好是在绘制完毕后（如果是使用双缓冲，则应该在交换缓冲后）立即调用。

大家可能会有这样的感觉：模板测试的设置是如此复杂，它可以实现的功能应该很多，肯定不止这样一个“限制像素的绘制范围”。事实上也是如此，不过现在我们暂时只讲这些。

其实，如果不需要绘制半透明效果，有时候可以用混合功能来代替模板测试。就绘制镜像这个例子来说，可以采用下面的步骤：

(1) 清除屏幕，在 glClearColor 中设置合适的值确保清除屏幕后像素的 Alpha 值为 0.0

(2) 关闭混合功能，绘制球体本身，设置合适的颜色（或者光照与材质）以确保所有被绘制的像素

的 Alpha 值为 0.0
(3) 绘制平面镜，设置合适的颜色（或者光照与材质）以确保所有被绘制的像素的 Alpha 值为 1.0
(4) 启用混合功能，用 GL_DST_ALPHA 作为源因子，GL_ONE_MINUS_DST_ALPHA 作为目标因子，这样就实现了只有原来 Alpha 为 1.0 的像素才能被修改，而原来 Alpha 为 0.0 的像素则保持不变。这时再绘制镜像物体，注意确保所有被绘制的像素的 Alpha 值为 1.0。
在有的 OpenGL 实现中，模板测试是软件实现的，而混合功能是硬件实现的，这时候可以考虑这样的代替方法以提高运行效率。但是并非所有的模板测试都可以用混合功能来代替，并且这样的代替显得不自然，复杂而且容易出错。
另外始终注意：使用混合来模拟时，即使某个像素原来的 Alpha 值为 0.0，以致于在绘制后其颜色不会有任何变化，但是这个像素的深度值有可能会被修改，而如果是使用模板测试，没有通过测试的像素其深度值不会发生任何变化。而且，模板测试和混合功能中，像素模板值的修改方式是不一样的。

4、深度测试

在本课的开头，已经简单的叙述了深度测试。这里是完整的内容。

深度测试需要深度缓冲区，跟模板测试需要模板缓冲区是类似的。如果使用 GLUT 工具包，可以在调用 glutInitDisplayMode 函数时在参数中加上 GLUT_DEPTH，这样来明确指定要求使用深度缓冲区。
深度测试和模板测试的实现原理很类似，都是在一个缓冲区保存像素的某个值，当需要进行测试时，将保存的值与另一个值进行比较，以确定是否通过测试。两者的区别在于：模板测试是设定一个值，在测试时用这个设定值与像素的“模板值”进行比较，而深度测试是根据顶点的空间坐标计算出深度，用这个深度与像素的“深度值”进行比较。也就是说，模板测试需要指定一个值作为比较参考，而深度测试中，这个比较用的参考值是 OpenGL 根据空间坐标自动计算的。

通过 glEnable/glDisable 函数可以启用或禁用深度测试。

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

至于通过测试的条件，同样有八种，与 Alpha 测试中的条件设置相同。条件设置是通过 glDepthFunc 函数完成的，默认值是 GL_LESS。

```
glDepthFunc(GL_LESS);
```

与模板测试相比，深度测试的应用要频繁得多。几乎所有的三维场景绘制都使用了深度测试。正因为这样，几乎所有的 OpenGL 实现都对深度测试提供了硬件支持，所以虽然两者的实现原理类似，但深度测试很可能会比模板测试快得多。当然了，两种测试在应用上很少有交集，一般不会出现使用一种测试去代替另一种测试的情况。
小结：
本次课程介绍了 OpenGL 所提供的四种测试，分别是剪裁测试、Alpha 测试、模板测试、深度测试。OpenGL 会对每个即将绘制的像素进行以上四种测试，每个像素只有通过一项测试后才会进入下一项测试，而只有通过所有测试的像素才会被绘制，没有通过测试的像素会被丢弃掉，不进行绘制。每种测试都可以单独的开启或者关闭，如果某项测试被关闭，则认为所有像素都可以顺利通过该项测试。
剪裁测试是指：只有位于指定矩形内部的像素才能通过测试。
Alpha 测试是指：只有 Alpha 值与设定值相比较，满足特定关系条件的像素才能通过测试。

模板测试是指：只有像素模板值与设定值相比较，满足特定关系条件的像素才能通过测试。
深度测试是指：只有像素深度值与新的深度值比较，满足特定关系条件的像素才能通过测试。
上面所说的特定关系条件可以是大于、小于、等于、大于等于、小于等于、不等于、始终通过、始终不通过这八种。
模板测试需要模板缓冲区，深度测试需要深度缓冲区。这些缓冲区都是在初始化 OpenGL 时指定的。如果使用 GLUT 工具包，则可以在 glutInitDisplayMode 函数中指定。无论是否开启深度测试，OpenGL 在像素被绘制时都会尝试修改像素的深度值；而只有开启模板测试时，OpenGL 才会尝试修改像素的模板值，模板测试被关闭时，OpenGL 在像素被绘制时也不会修改像素的模板值。
利用这些测试操作可以控制像素被绘制或不被绘制，从而实现一些特殊效果。利用混合功能可以实现半透明，通过设置也可以实现完全透明，因而可以模拟像素颜色的绘制或不绘制。但注意，这里仅仅是颜色的模拟。OpenGL 可以为像素保存颜色、深度值和模板值，利用混合实现透明时，像素颜色不发生变化，但深度值则会可能变化，模板值受 glStencilFunc 函数中第三个参数影响；利用测试操作实现透明时，像素颜色不发生变化，深度值也不发生变化，模板值受 glStencilFunc 函数中前两个参数影响。
此外，修正了第十课、第十一课中的一个函数中的 bug。在 grab 函数中，应该在最开头加上一句 glReadBuffer(GL_FRONT);以保证读取到的内容正好就是显示的内容。

因为论坛支持附件了，我会把程序源代码和所使用的图片上传到附件里，方便大家下载。

点击下载附件

： http://file.pfan.cn/down/bbs/7/20071007731.zip