电子科技大学信息与软件工程学院

实 验 指 导 书

（实验）课程名称 图形与动画 II

电子科技大学教务处制表

目　　录

实验二 OpenGL 基础实验（二）

一、实验目的

1. 掌握投影变换的方法；
2. 掌握照相机移动方法；
3. 掌握颜色以及着色模式的设置方法；
4. 掌握光照的基础知识和设置方法；
5. 掌握材质的设置方法；
6. 掌握纹理贴图的方法。

二、实验原理

（一） 几何转换

1. 在场景中设置物体位置

1.1 移动物体

glTranslatef（Glfloat x, GLfloat y, GLfloat z）；

参数分别表示沿 x、y 和 z 方向移动的数量。

1.2 旋转物体

glRotatef（Glfloat angle, Glfloat x, GLfloat y, GLfloat z）；

物体绕着由 x、y 和 z 参数所指定的向量旋转 angle 度（逆时针方向）。

1.3 缩放物体

glScalef（Glfloat x, GLfloat y, GLfloat z）；

参数 x、y 和 z 指定了物体沿各个方向缩放倍数。

2. 透视投影

之前介绍的窗口刷新函数 changeSize 利用正投影变换来剪裁投影空间，接下来看看如何利用透视投影来剪裁。

OpenGL 提供函数 gluPerspective 来实现透视投影：

void **gluPerspectiv**(GLdouble fovy, GLdouble aspect, GLdouble zNear, Gldouble zfar);

gluPerspective 函数的参数：

- fovy - 垂直方向的视野角度；
- aspect- 裁剪平面高度和宽度的比；

- zNear 和 zfar - 近端和远端裁剪平面之间的距离。

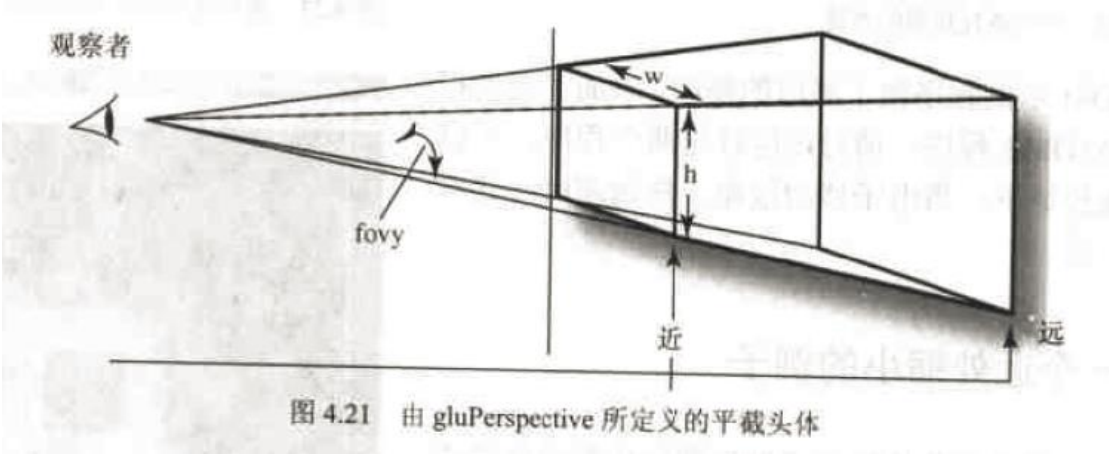


图 4.21 由 gluPerspective 所定义的平截头体

图 1

3. 利用透视变换实现窗口刷新

利用透视投影来实现窗口刷新函数 changeSize:

```
// Change viewing volume and viewport.  Called when window is resized
void ChangeSize(GLsizei w, GLsizei h)
    {
    GLfloat fAspect;

    // Prevent a divide by zero
    if(h == 0)
        h = 1;

    // Set Viewport to window dimensions
    glViewport(0, 0, w, h);

    fAspect = (GLfloat)w/(GLfloat)h;

    // Reset coordinate system
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    // Produce the perspective projection
    gluPerspective(60.0f, fAspect, 1.0, 400.0);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    }
```

图 2

4. 移动镜头

一般情况下实现场景变换有两种方法：

1) 移动场景中物体 – 该方法比较简单,使用 glTranslatef 与 glRotatef 配合即可,但一般只在简单场景和单角色的情况下使用,而且角色的各种计算(如实时坐标、碰撞)不好

实现，所以不推荐使用；

2) 移动视点（镜头） – 该方法就非常灵活，它对场景和角色的状态未做任何操作，一般只要修改观察点位置、角度就可以实现视角变化视觉效果。OpenGL 提供了一个函数 gluLookAt 提供一个简单直观的方式来设置镜头位置和方向：

void **gluLookAt**(GLdouble eyex,GLdouble eyey,GLdouble eyez,
GLdouble centerx,GLdouble centery,GLdouble centerz,
GLdouble upx,GLdouble upy,GLdouble upz);

一般而言它有三组参数，每组由三个浮点型值组成：

- 第一组 eyex, eyey,eyez 镜头位置在世界坐标的位置；
- 第二组 centerx,centery,centerz 目视点位置，决定视线的方向；
- 第三组 upx,upy,upz 镜头向上的方向，一般设置为(0.0, 1.0, 0.0)，意思是镜头是没有倾斜的。如果你希望镜头倾斜你可以更改这组值。例如，改成倒视视觉是(0.0, -1.0, 0.0)。

如果我们用向量 **e** 表示向量镜头位置在世界坐标的位置，用向量 **c** 表示视线的方向，用向量 **u** 表示镜头向上的方向，那么它们三者关系如图 3 所示。

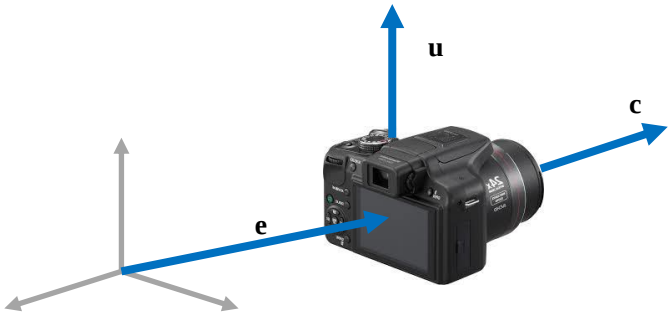


图 3

5. 移动镜头的应用

下面来看一个更有趣的 GLUT 应用。

本节我们会绘制一个雪人世界，并直接用按键移动镜头。向左和向右键会在 XZ 切面围绕 Y 轴旋转镜头。反之，向上和向下键会在当前方向向前向后移动镜头。

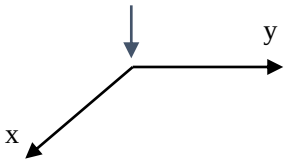
实现代码已经在适当地方标上注释。

5.1 全局变量

首先我们要一些全局变量来保存镜头参数（如图 4 所示）。这些变量会保存了镜头位置和目标方向的向量。我们还需要保存角度。由于 y 是常量，所以不用保存。

- angle: y 轴上旋转的角度。该变量是旋转镜头用；

- x,z: XZ 平面的镜头位置；
- lx, lz: 定义我们视线的向量；



以上变量的赋值如下:

```
// angle of rotation for the camera direction
float angle=0.0;
// actual vector representing the camera's direction
float lx=0.0f,lz=-1.0f;
// XZ position of the camera
float x=0.0f,z=5.0f;
```

图 4

5.2 绘制雪人函数 drawSnowMan

drawSnowMan 函数包含所有的绘制雪人世界的命令（如图 5 所示）。

```
void drawSnowMan() {

    glColor3f(1.0f, 1.0f, 1.0f);

// Draw Body
    glTranslatef(0.0f ,0.75f, 0.0f);
    glutSolidSphere(0.75f,20,20);

// Draw Head
    glTranslatef(0.0f, 1.0f, 0.0f);
    glutSolidSphere(0.25f,20,20);

// Draw Eyes
    glPushMatrix();
    glColor3f(0.0f,0.0f,0.0f);
    glTranslatef(0.05f, 0.10f, 0.18f);
    glutSolidSphere(0.05f,10,10);
    glTranslatef(-0.1f, 0.0f, 0.0f);
    glutSolidSphere(0.05f,10,10);
    glPopMatrix();

// Draw Nose
    glColor3f(1.0f, 0.5f , 0.5f);
    glRotatef(0.0f,1.0f, 0.0f, 0.0f);
    glutSolidCone(0.08f,0.5f,10,2);
}
```

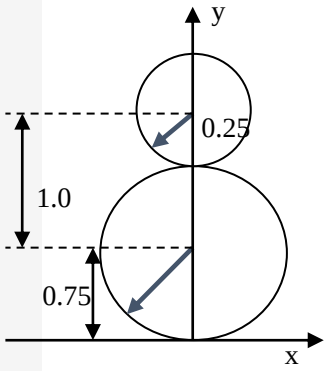


图 5

5.3 场景绘制函数 renderScene

如**图 6** 所示渲染函数的实现代码:

```
void renderScene(void) {

    // Clear Color and Depth Buffers

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    // Reset transformations
    glLoadIdentity();
    // Set the camera
    gluLookAt(    x, 1.0f, z,
            x+lx, 1.0f,  z+lz,
            0.0f, 1.0f,  0.0f);

        // Draw ground
    glColor3f(0.9f, 0.9f, 0.9f);
    glBegin(GL_QUADS);
        glVertex3f(-100.0f, 0.0f, -100.0f);
        glVertex3f(-100.0f, 0.0f,  100.0f);
        glVertex3f( 100.0f, 0.0f,  100.0f);
        glVertex3f( 100.0f, 0.0f, -100.0f);
    glEnd();
```

```
        // Draw 36 SnowMen
    for(int i = -3; i < 3; i++)
        for(int j=-3; j < 3; j++) {
            glPushMatrix();
            glTranslatef(i*10.0,0,j * 10.0);
            drawSnowMan();
            glPopMatrix();
        }

    glutSwapBuffers();
}
```

图 6

两层嵌套的 for 循环总共调用了 36 次函数 drawSonwMan 在不同位置绘制了 36 个雪人。

5.4 处理键盘事件

现在开始处理箭头键(上下左右)。我们用左右箭头键来旋转镜头，例如更改向量来定义视线。上下键用来移动镜头的位置-沿着视线方向前进或者后退。

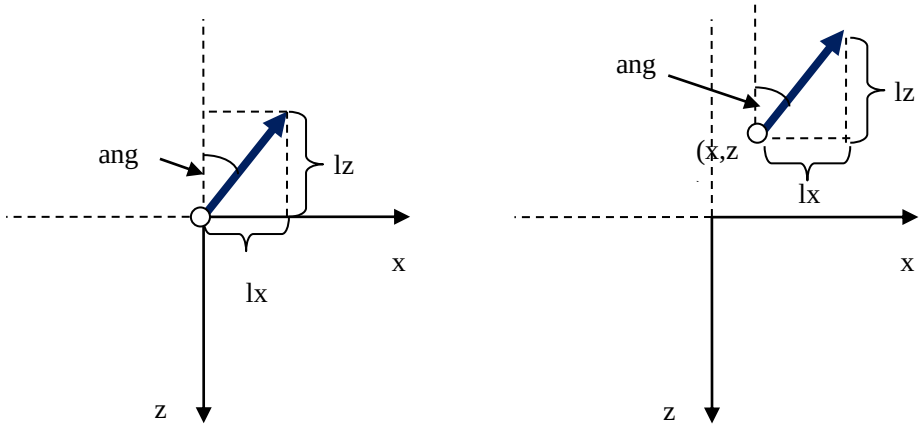


图 7

5.1.1 改变视线方向

当用户按左右键时，视线角度变量也会跟着改变。随着角度值的改变，程序会重新计算出视线向量的 lx 和 lz 相应的合适的值。留意到我们现在只是在 XZ 平面移动，所以我们不用改变视觉向量的 ly 坐标。新的 lx 和 lz 值会映射到单一的 XZ 平面的圆圈上。于是，下面得出了计算角度 ang 和新的 lx,lz 值的公式(如图 7 所示):

$$lx = \sin(ang);$$

$$lz = -\cos(ang);$$

跟我们把极坐标转换成平面坐标一样。

注意，当更新 lx 和 lz 时镜头是不移动的，镜头位置不变，只有目视点改变。

目视点坐标为：

$$x + lx$$

$$z + lz$$

回过头来看看**图 6** 中的 gluLookAt 函数，参数：

- x, 1.0f, z 指定了镜头坐标；
- x+lx, 1.0f, z+lz 指定了视线方向；

5.4.2 改变镜头位置

我们还想沿着视线移动镜头。为了达到这个效果我们需要分别在按上/下键的时候加/减一个粒度的视觉向量到当前位置。例如，移动镜头向前时，x 和 z 的计算公式如下:

x = x + lx * 粒度

z = z + lz * 粒度

这里的粒度是指一个合适的速率。我们知道 lx 和 lz 是一个单一向量(前面提及它是一个单位周期中的点)，因此如果粒度值保持在一个常量，速率便会维持在一个常量变化的感觉。增加粒度我们就移动得快一点，也就是说，我们会在每一帧移动得更远。

5.4.3 完整的键盘响应函数

```
void processSpecialKeys(int key, int xx, int yy) {

    float fraction = 0.1f;

    switch (key) {
        case GLUT_KEY_LEFT :
            angle -= 0.01f;
            lx = sin(angle);
            lz = -cos(angle);
            break;
        case GLUT_KEY_RIGHT :
            angle += 0.01f;
            lx = sin(angle);
            lz = -cos(angle);
            break;
        case GLUT_KEY_UP :
            x += lx * fraction;
            z += lz * fraction;
            break;
        case GLUT_KEY_DOWN :
            x -= lx * fraction;
            z -= lz * fraction;
            break;
    }
}
```

图 8

5.5 主函数

配合着以上代码的更改,main 函数里面也增加一些代码。唯一的变化是开启了深度测试特性（如图 9 所示）。

```
int main(int argc, char **argv) {

    // init GLUT and create window

    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100,100);
    glutInitWindowSize(320,320);
    glutCreateWindow("Lighthouse3D - GLUT Tutorial");

    // register callbacks
    glutDisplayFunc(renderScene);
    glutReshapeFunc(changeSize);
    glutIdleFunc(renderScene);
    glutKeyboardFunc(processNormalKeys);
    glutSpecialFunc(processSpecialKeys);

    // OpenGL init
    glEnable(GL_DEPTH_TEST);

    // enter GLUT event processing cycle
    glutMainLoop();

    return 1;
```

图 9

6. 本节学习函数

目标	实现函数	目标	实现函数
在场景中建立你的位置	gluLookAt	建立透视转换	gluPerspective
在场景中设置物体的位置	glTranslate/glRotate	执行矩阵转换	glLoadMatrix/glMutMatrix
伸缩物体	glScale	使用照相机在场景中移动	gluLookAt

（二）颜色

1. 定义颜色方式

OpenGL 通过制定红、绿、蓝成分的强度来指定一种颜色。

glColor<x><t>(red, green, blue, alpha);

- <x>表示参数的数量，可以是 3 或 4；
- <t>表示参数的数据类型，可以是 b,d,f,i,s,ub,ui 或 us,分别表示 byte, double, float, integer, short, unsigned byte, unsigned integer。

2. 着色模式（shading model）

glColor 函数用于设置当前的绘图颜色，在这条命令之后所绘制的所有物体用最后一次指定的颜色绘制。当前，所有绘图例子都是通过 glColor 指定颜色后，用该颜色对整个物体绘制。但是，如果一个图元每个顶点指定了不同颜色，图元内部如何着色？这取决于**着色模式**（shading model）。

着色可以简单地定义为图元指定颜色渲染方式。假设我们绘制三角形，为三角形每个顶点设置了不同的颜色，那么三角形内部着色模式默认采用一种颜色到下一种颜色的平滑过渡这种模式（GL_SMOOTH），如**图 10** 所示。

```
// Called to draw scene
void RenderScene(void)
{
    // Clear the window with current clearing color
    glClear(GL_COLOR_BUFFER_BIT);

    // Enable smooth shading
    glShadeModel(GL_SMOOTH);

    // Draw the triangle
    glBegin(GL_TRIANGLES);
        // Red Apex
        glColor3ub((GLubyte)255, (GLubyte)0, (GLubyte)0);
        glVertex3f(0.0f, 200.0f, 0.0f);

        // Green on the right bottom corner
        glColor3ub((GLubyte)0, (GLubyte)255, (GLubyte)0);
        glVertex3f(200.0f, -70.0f, 0.0f);

        // Blue on the left bottom corner
        glColor3ub((GLubyte)0, (GLubyte)0, (GLubyte)255);
        glVertex3f(-200.0f, -70.0f, 0.0f);
    glEnd();

    // Flush drawing commands
    glutSwapBuffers();
}
```

图 10

可以使用 glShadeModel 函数设置另外一种着色模式 – GL_FLAT，它表示单调着色。使用该着色方式，图元内部颜色采用最后一个顶点所指定的额颜色。但是 GL_POLYGON 图元例外，它的内部颜色是第一个顶点的颜色。

（三）光照

1. 光照种类

物体自身不会发光，它一般受到 3 种不同类型的光的照射：

- 环境光（ambient）；
- 散射/反射光（diffuse）；
- 镜面光（specular）；

1.1 环境光（ambient）

环境光并不来自任何特定的方向，它来自某个光源，但光线却是在场景中四处反射。由环境光所照射的物体在所有方向上的所有表面都是均匀照亮的（如**图 11** 所示）。

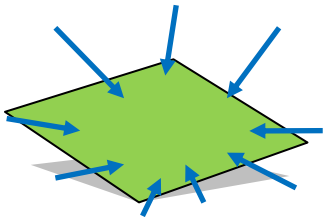


图 11

1.2 散射/反射光（diffuse）

散射光来自于某一特定方向，但它均匀地在一个表面反射开来。虽然光是均匀反射的，但受到光线直接照射的物体表面还是要亮一些（如**图 12** 所示）。

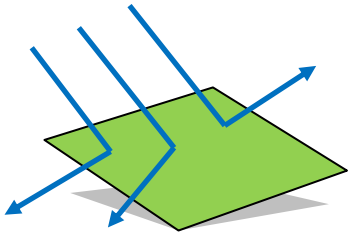


图 12

1.3 镜面光（specular）

光线来自于某一特定方向，反射角度也是沿着某一特定方向（如**图 13** 所示）。高强度镜面光趋向于在它所照射的表面形成一个亮点，称为光斑。

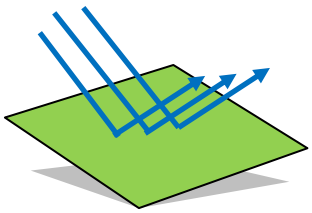


图 13

2. 添加光源

接下来看看如何在 OpenGL 中添加光源。

2.1 启用光源

为了告诉 OpenGL 使用光源，需要调用 glEnable 函数，如下所示：

```
glEnable(GL_LIGHTING);
```

这个调用告诉 OpenGL 在确定场景中每个顶点的颜色时，使用材料属性和光照参数。

2.2 设置光照模型

启用光照计算后，开始设置光照模型。通过 glLight 函数设置影响光照模型的 3 个参数的值。

2.2.1 设置光源参数

设置光源的强度、位置，如**图 14** 所示：

```
GLfloat LightAmbient[]=     { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightDiffuse[]=     { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightPosition[]=    { 0.0f, 0.0f, 2.0f, 1.0f };
```

图 14

2.2.2 设置并启用光照

下面这段代码启用已设置好的亮白色的光：

```
glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);      // Setup The Ambient Light
glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);      // Setup The Diffuse Light
glLightfv(GL_LIGHT1, GL_POSITION,LightPosition);     // Position The Light
glEnable(GL_LIGHT1);                                 // Enable Light One
```

图 15

3. 绘制一个旋转正方体

3.1 全局变量

```
#include <glut.h>
#include <stdlib.h>

GLfloat xrot;               // X Rotation
GLfloat yrot;               // Y Rotation
GLfloat xspeed;             // X Rotation Speed
GLfloat yspeed;             // Y Rotation Speed
GLfloat z=-5.0f;            // Depth Into The Screen
```

图 16

3.2 主函数

```
int main(int argc, char **argv) {

    // init GLUT and create window
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100,100);
    glutInitWindowSize(640,640);
    glutCreateWindow("Rotate Cube");

    // register callbacks
    glutDisplayFunc(renderScene);
    glutIdleFunc(renderScene);
    glutReshapeFunc(changeSize);
    glutSpecialFunc(processSpecialKeys);

    InitGL();

    // enter GLUT event processing loop
    glutMainLoop();

    return 0;
}
```

图 17

3.3 changeSize 函数

```
void changeSize(int w, int h) {

    // Prevent a divide by zero, when window is too short
    // (you cant make a window of zero width).
    if (h == 0)
        h = 1;

    float ratio =  w * 1.0 / h;

    // Use the Projection Matrix
    glMatrixMode(GL_PROJECTION);

    // Reset Matrix
    glLoadIdentity();

    // Set the viewport to be the entire window
    glViewport(0, 0, w, h);

    // Set the correct perspective.
    gluPerspective(45,ratio,1,100);

    // Get Back to the Modelview
    glMatrixMode(GL_MODELVIEW);
}
```

图 18

3.4 绘图初始化函数

```
int InitGL(GLvoid)                                      // All Setup For OpenGL Goes Here
{
    glShadeModel(GL_SMOOTH);                            // Enable Smooth Shading
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);               // Black Background
    glClearDepth(1.0f);                                 // Depth Buffer Setup
    glEnable(GL_DEPTH_TEST);                            // Enables Depth Testing
    glDepthFunc(GL_LEQUAL);                             // The Type Of Depth Testing To Do
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);  // Really Nice Perspective Calculations

    return true;                                            // Initialization Went OK
}
```

图 19

3.5 绘制函数

```
void renderScene(void) {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
    glLoadIdentity();                                   // Reset The View
    glTranslatef(0.0f,0.0f,z);

    glRotatef(xrot,1.0f,0.0f,0.0f);
    glRotatef(yrot,0.0f,1.0f,0.0f);

    glColor4f(1.0f,0.0f,0.0f,1.0f);
    glutSolidCube(1.0);

    glutSwapBuffers() ;

    xrot += xspeed;
    yrot += yspeed;
}
```

图 20

3.6 键盘响应函数

```
void processSpecialKeys(int key, int x, int y) {
    if (key ==  GLUT_KEY_PAGE_UP){
        z-=0.02f;
    }
    if (key == GLUT_KEY_PAGE_DOWN){
        z+=0.02f;
    }
    if (key == GLUT_KEY_UP){
        xspeed-=0.01f;
    }
    if (key == GLUT_KEY_DOWN){
        xspeed+=0.01f;
    }
    if (key == GLUT_KEY_LEFT){
        yspeed-=0.01f;
    }
    if (key == GLUT_KEY_RIGHT){
        yspeed-=0.01f;
    }

}
```

图 21

正确执行后，正方体绘制如图 22 所示，没有立体效果，这是因为没有加入光照：

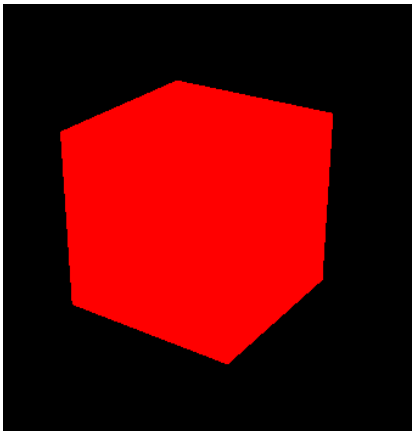


图 22

4. 加入光照

4.1 启用光源

通过按键“*l*”控制光照模式的开启，添加如下函数：

```
void processNormalKeys(unsigned char key, int x, int y) {
    switch(key)
    {
        case 27:                                    /** Esc按键按下后退出程序 */
            exit(0);
            break;
        case 'l':                                   /** 按键L 光源开关*/
            light=!light;
            light? glEnable(GL_LIGHTING):glDisable(GL_LIGHTING);
            break;
    }
}
```

图 23

同时在主函数中注册该函数：

glutKeyboardFunc(processNormalKeys);

4.2 设置光照模型

4.2.1. 设置光源参数

在程序头部设置光源参数，如**图 14** 所示。

4.2.2. 设置并启用光照

在初始化绘图函数 IniGL 中添加**图 15** 中代码。

正常显示后如所示：

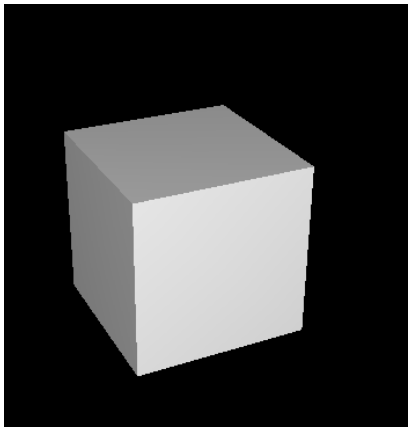


图 24

大家注意到此时虽然正方体具有了立体的效果，但是设置正方体的颜色不再起任何作用，它始终为白色。这是因为我们未设置物体材质。

（四）材质

1. 材质的定义

一个物体为什么表现出颜色？这是由它的材质决定的，它的材质决定了它所反射的光的波长。一个红色的物体是由于它反射了光线中的红色光，并吸收了其他部分的光。

比如：一个光源的 RGB 值为（0.5,0.5,0.5），一个物体的反射属性为（0.5,1.0,0.5），那么物体反射光最后为：

(0.5*0.5, 0.5*1.0, 0.5*0.5) = (0.25, 0.5, 0.25)

也就是说光源中红色成分一半被反射，绿色成分完全被反射，蓝色成分一半被反射。

2. 设置材质的属性

2.1 方法一：

添加如下参数：

```
GLfloat gray[] = {0.9f, 0.0f, 0.0f, 1.0f};
```

该变量指定了物体表面的材质，如上例所示，物体反射光源中 90%红光，吸收所有的绿光、蓝光。

接下来通过 glMaterialfv 函数设置**材质属性**，在 InitGL 函数中添加：

```
glMaterialfv(GL_FRONT, GL_DIFFUSE,gray);
```

- 第一个参数指定了材质属性作用于正面、背面还是双面（GL_FRONT、GL_BACK 或 GL_FRONT_AND_BACK）；
- 第二个参数指定哪个类型的光将被设置，此处设置散射光；
- 第三个参数指定了反射系数；

最后绘制结果如**图 25** 所示：

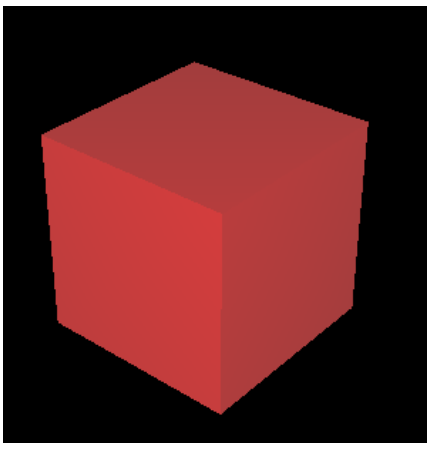


图 25

2.2 方法二：

一种更方便的方法叫做**颜色追踪**。该方法通过 glColor 来设置材料属性。为了启用颜色追踪，需要调用 glColorMaterial 函数：

```
glColorMaterial (GLenum face,GLenum mode);
```

- face的取值: GL_FRONT, GL_BACK, GL_FRONT_AND_BACK ;
- mode 的取值：GL_AMBIENT, GL_DIFFUSE, GL_AMBIENT_AND_DIFFUSE, GL_SPECULAR, GL_EMISSION;

例如：为了设置多边形正面的散射光属性以便追踪glColor所设置的颜色，调用：

glColorMaterial(GL_FRONT, GL_ DIFFUSE);

要使用 glColorMaterial，必须启用 GL_COLOR_MATERIAL：

glEnable(GL_COLOR_MATERIAL);

接着函数遵循glColor所设置的值来指定材料参数。

注意：glColorMaterial必须出现在glEnable(GL_COLOR_MATERIAL);之前。

在此例中，设置正方体为蓝色（如**图 26**所示）：

glColor(0.0f,0.0f,0.9f,1.0f);

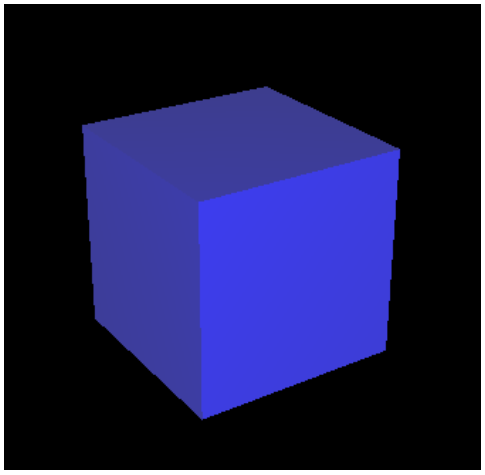


图 26

（五）纹理贴图

为了让图形更真实，我们可以将图片贴在物体表面，这一过程称为纹理贴图（Texture Map）。下面看看如何实现纹理贴图。

1. 载入纹理

将纹理贴图应用到几何图形的第一步是将纹理载入到内存中。载入纹理关键就是读入一张图片。

早期 OpenGL 程序普遍采用 glaux 库读入图片，该方法一直以来受到反对。这里我们学习使用 SOIL 库，SOIL 库使用更方便更受欢迎。使用 SOIL 库第一步确保 SOIL.h 和 SOIL.lib 文件和你编写的 OpenGL 源文件放在一起。此外，在源代码开头处，添加如下代码：

#include “SOIL.h”

另外，项目需要链接到 SOIL.lib。在 vs2010 下配置如下：

1) 点击**项目** – 选择“**xx 项目属性**”；
2) 选择**配置属性** – **链接器** – **输入；**
3) 在**附加依赖项**中添加“SOIL.lib”;

下面可以载入图像并生成纹理，如**图 27 红色方框**所示：

```
int LoadGLTextures()                                    // Load Bitmaps And Convert To Textures
{
    for (int i = 0; i < 2; i++)
    /* load an image file directly as a new OpenGL texture */
    texture[i] = SOIL_load_OGL_texture
        (
        "Data/glass.bmp",
        SOIL_LOAD_AUTO,
        SOIL_CREATE_NEW_ID,
        SOIL_FLAG_INVERT_Y
        );

    if(texture[0] == 0)
        return false;

        // Create Nearest Filtered Texture
        glBindTexture(GL_TEXTURE_2D, texture[0]);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_NEAREST);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_NEAREST);

    if(texture[1] == 0)
        return false;
        // Create Linear Filtered Texture
        glBindTexture(GL_TEXTURE_2D, texture[1]);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MAG_FILTER,GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D,GL_TEXTURE_MIN_FILTER,GL_LINEAR);

    return true;                                        // Return Success
}
```

图 27

整个载入图像过程都被封装在函数 SOIL_load_OGL_texture 中，它的第一个参数指明了图片路径和名字，第二个参数 SOIL_LOAD_AUTO 指原样载入图片，第三个参数 SOIL_CREATE_NEW_ID 告诉 SOIL 为当前载入的纹理创建一个 ID。

2. 将纹理贴图到几何图形

载入纹理并启用纹理功能后就可以把这个纹理应用到相应图元。但是必须告诉 OpenGL 如何把纹理贴图到几何图形。

纹理具有自己的坐标，纹理坐标为 s、t、r 和 q（类似于几何图形坐标 x、y、z 和 w）。纹理坐标为浮点数，范围从 0.0 到 1.0。

纹理贴图到几何图形就是要建立起纹理坐标和几何图形坐标的对应关系，如**图 28** 所示：

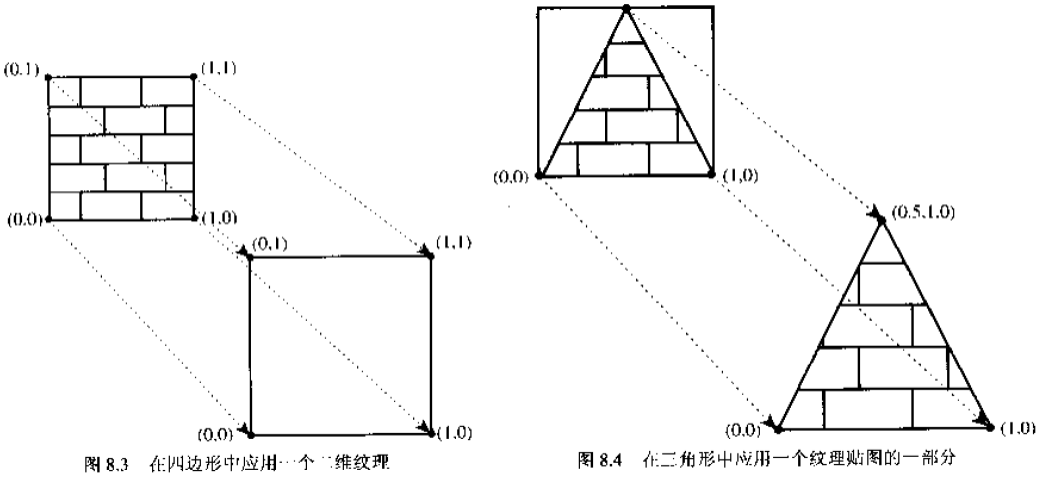


图 8.3 在四边形中应用一个二维纹理　　图 8.4 在三角形中应用一个纹理贴图的一部分

图 28

OpenGL 中有两种方法为几何图形的顶点(Vertex)指定纹理坐标：

1) 由人工给每个顶点分配坐标。可以通过函数 glTexCord*()来完成。

2) 由 OpenGL 自动为每个顶点分配坐标。这个任务由函数 glTexGen*()来完成。

在指定了纹理坐标和物体几何坐标对应关系后，需要指定采用什么模式生成纹理。我们先以人工方式给每个顶点分配纹理坐标为例讲解。

2.1 指定纹理坐标

我们根据一个简单的例子来看看如何人工设置纹理坐标。如**图 29** 所示，我们需要将正方形纹理（蓝色）贴到正方形图元上（绿色）。

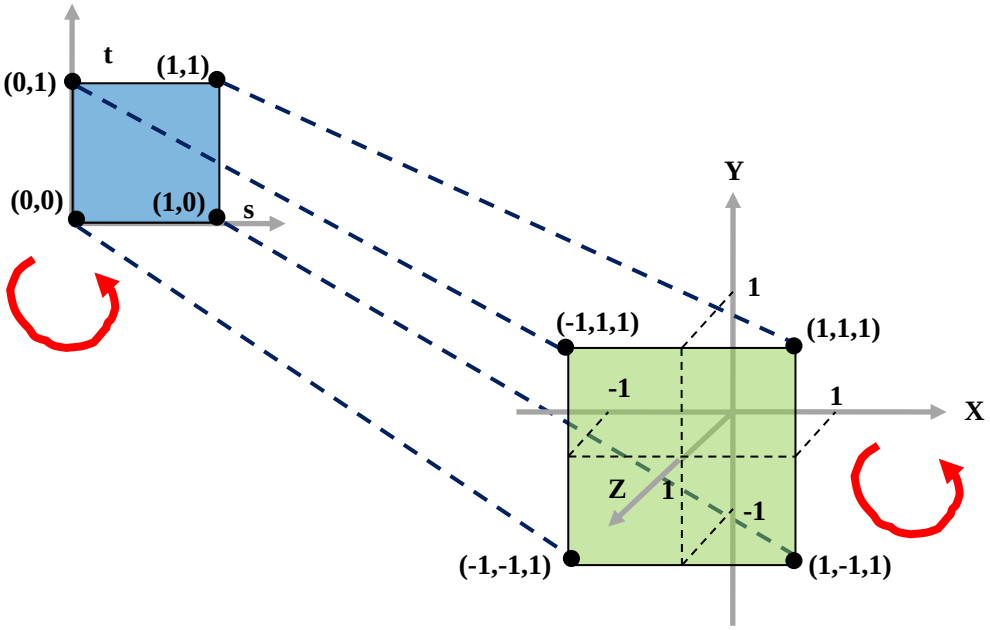


图 29

如**图 29** 所示，要建立纹理（蓝色）和正方形（绿色）顶点对应关系，需要执行如下操作（**图 30** 所示）：

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

图 30

除了指定了坐标的对应关系，还需要指定几何平面法线方向。

接下来我们回过头来看看在载入纹理后，如何设置纹理环境。如**图 27** 绿色框所示，用到了 **glTexParameterfv** 函数。

由于纹理大小不一定和几何图形大小相同，因此在几何图形上进行纹理贴图时，纹理图像会根据几何图形大小而进行拉伸或者收缩，这个过程称为纹理过滤（Texture Filtering）。在 **glTexParameterfv** 函数中第二个参数 GL_TEXTURE_MAG_FILTER 和 GL_TEXTURE_MIN_FILTER 就分别设置放大和缩小过滤器；**glTexParameterfv** 函数中第三个参数 GL_NEAREST 和 GL_LINEAR 分别对应了在执行放大、缩小纹理时采用的方法，分别为最邻近过滤和线性过滤。最邻近过滤是最简单和最快速的方法，但是缺点是纹理被拉伸到特别大时会出现大片斑驳状像素；线性过滤计算开销大些，不过当今的高速硬件上线性过滤带来的额外开销可以忽略不计。线性过滤是把纹理坐标周围的纹理单元的加权平均值应用到对应纹理坐标上。

完整代码如下：

- **全局变量：**

```
#include <glut.h>
#include "SOIL.h"
#include <stdio.h>
#include <stdlib.h>


bool    light;              // Lighting ON/OFF

GLfloat xrot;               // X Rotation
GLfloat yrot;               // Y Rotation
GLfloat xspeed;             // X Rotation Speed
GLfloat yspeed;             // Y Rotation Speed
GLfloat z=-5.0f;            // Depth Into The Screen

GLfloat LightAmbient[]=     { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat LightDiffuse[]=     { 1.0f, 1.0f, 1.0f, 1.0f };
GLfloat LightPosition[]=    { 0.0f, 0.0f, 2.0f, 1.0f };

GLuint  filter;             // Which Filter To Use
GLuint  texture[2];         // Storage For 2 Textures
```

图 31

- **主函数：**

```
int main(int argc, char **argv) {

    // init GLUT and create window
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA);
    glutInitWindowPosition(100,100);
    glutInitWindowSize(640,640);
    glutCreateWindow("Texture");

    // register callbacks
    glutDisplayFunc(renderScene);
    glutIdleFunc(renderScene);
    glutReshapeFunc(changeSize);
    glutKeyboardFunc(processNormalKeys);
    glutSpecialFunc(processSpecialKeys);

    if(!InitGL()){
        printf("Initialization Failed.");
        return false;
    }
     // enter GLUT event processing loop
    glutMainLoop();

    return 0;
}
```

图 32

所涉及的函数 processSpecialKeys 如**图 21** 所示。

- **绘制函数：**

```
void renderScene(void) {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); // Clear The Screen And The Depth Buffer
    glLoadIdentity();                                   // Reset The View
    glTranslatef(0.0f,0.0f,z);

    glRotatef(xrot,1.0f,0.0f,0.0f);
    glRotatef(yrot,0.0f,1.0f,0.0f);

    glBindTexture(GL_TEXTURE_2D, texture[filter]);

    glBegin(GL_QUADS);
        // Front Face
        glNormal3f( 0.0f, 0.0f, 1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
        // Back Face
        glNormal3f( 0.0f, 0.0f,-1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
        // Top Face
        glNormal3f( 0.0f, 1.0f, 0.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
        // Bottom Face
        glNormal3f( 0.0f,-1.0f, 0.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
        // Right face
        glNormal3f( 1.0f, 0.0f, 0.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f( 1.0f, -1.0f, -1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f( 1.0f,  1.0f, -1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f( 1.0f,  1.0f,  1.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f( 1.0f, -1.0f,  1.0f);
        // Left Face
        glNormal3f(-1.0f, 0.0f, 0.0f);
        glTexCoord2f(0.0f, 0.0f); glVertex3f(-1.0f, -1.0f, -1.0f);
        glTexCoord2f(1.0f, 0.0f); glVertex3f(-1.0f, -1.0f,  1.0f);
        glTexCoord2f(1.0f, 1.0f); glVertex3f(-1.0f,  1.0f,  1.0f);
        glTexCoord2f(0.0f, 1.0f); glVertex3f(-1.0f,  1.0f, -1.0f);
    glEnd();

    glutSwapBuffers() ;

    xrot+=xspeed;
    yrot+=yspeed;
}
```

图 33

- **绘图初始化函数：**

```
int InitGL(GLvoid)                                      // All Setup For OpenGL Goes Here
{
    if (!LoadGLTextures())                              // Jump To Texture Loading Routine
    {
        return false;                                   // If Texture Didn't Load Return FALSE
    }

    glEnable(GL_TEXTURE_2D);                            // Enable Texture Mapping
    glShadeModel(GL_SMOOTH);                            // Enable Smooth Shading
    glClearColor(0.0f, 0.0f, 0.0f, 0.5f);               // Black Background
    glClearDepth(1.0f);                                 // Depth Buffer Setup
    glEnable(GL_DEPTH_TEST);                            // Enables Depth Testing
    glDepthFunc(GL_LEQUAL);                             // The Type Of Depth Testing To Do
    glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_NICEST);  // Really Nice Perspective Calculations

    glLightfv(GL_LIGHT1, GL_AMBIENT, LightAmbient);     // Setup The Ambient Light
    glLightfv(GL_LIGHT1, GL_DIFFUSE, LightDiffuse);     // Setup The Diffuse Light
    glLightfv(GL_LIGHT1, GL_POSITION,LightPosition);    // Position The Light
    glEnable(GL_LIGHT1);                                // Enable Light One

    return true;                                            // Initialization Went OK
}
```

图 34

LoadGLTextures 函数如**图 27** 所示。

- **键盘响应函数：**

```
void processNormalKeys(unsigned char key, int x, int y) {
     switch(key)
     {
         case 27:                                       /** Esc按键按下后退出程序 */
             exit(0);
             break;
         case 'l':                                      /** 按键L 光源开关*/
             light = !light;
             light? glEnable(GL_LIGHTING):glDisable(GL_LIGHTING);
             break;
         case 'f':                                      /** 选择滤波方式*/
             filter += 1;
             if (filter > 1) filter = 0;
             printf("%d",filter);
             break;
     }
}
```

图 35

2.2 自动生成纹理坐标

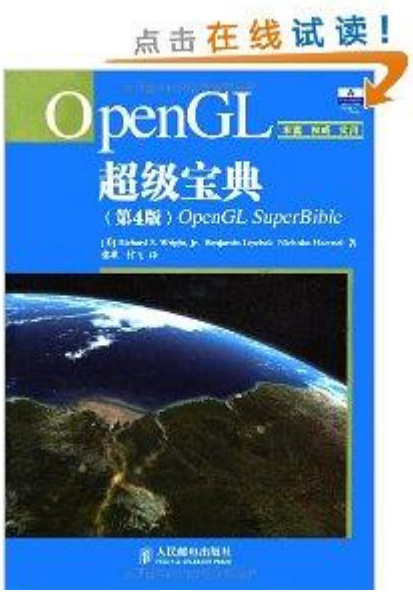
略。

3. 纹理环境高级设置

略。

参考文献

1. 《OpenGL超级宝典》（俗称“蓝宝书bluebook”）第四、五、八章



三、实验内容

1. 利用透视投影视线 changeSize 函数；
2. 实现移动镜头的应用 - 绘制一个雪人世界；
3. 绘制一个三角形，顶点设置不同颜色，对比两种着色模式 - GL_SMOOTH 和 GL_FLAT；
4. 绘制一个旋转的立方体，并向场景中添加光源：环境光、散射光；
5. 采用两种不同方法为在上例中给立方体设置材质属性；
6. 为雪人世界添加光照、设置材质属性；
7. 为 5）中的立方体添加纹理；

四、实验步骤

1. 明确项目需求；
2. 编写代码；
3. 编译代码；
4. 测试程序；
5. 根据测试结果对程序进行调试改进。