湖南科技大学课程教案
（章节、专题首页）

授课教师：王志喜　　职称：副教授　　单位：计算机科学与工程学院

课程名称	计算机图形图像技术
章节、专题	OpenGL的真实感图形
教学目标及基本要求	掌握OpenGL中的光照处理方法等真实感绘制方法。
教学重点	OpenGL的基本光照处理，OpenGL的融合
教学难点	OpenGL的基本光照处理
教学内容与时间分配	（1）OpenGL的基本光照处理（0.7课时） （2）OpenGL光照处理的例子（0.9课时） （3）融合（0.4课时） 共计2课时。
习　　题	第8.7.1节（基础知识题）和8.7.2节（程序设计题）。

第8章　OpenGL的真实感图形

8.1　光照处理

8.1.1　光照处理的基本概念

OpenGL的光是由红、绿、蓝组成的。光源的颜色由其所发出的红绿蓝颜色的数量决定，材料的属性用反射的红绿蓝颜色的百分比表示。

OpenGL的光可来自多个光源，每个光源可以单独控制开关。有的光源来自某个特定的方向或位置，有的光源分散于整个场景。

1. 光源的构成

光源由环境光成分（简称为泛光）、漫反射成分和镜面反射成分三部分构成。

- 环境光成分。来自环境，在各方向均匀分布。
- 漫反射成分。来自某一特定方向，一旦照射到表面上，无论在何处观察，亮度相同。
- 镜面反射成分。来自某一特定方向，以一特定方向离开。

2. 材料颜色

- 取决于材料对红绿蓝光的反射能力，包括环境光反射（简称为泛射）、漫反射和镜面反射。
- 材料的环境光反射与光源的环境光成分相对应。
- 材料的漫反射与光源的漫反射成分相对应。
- 材料的镜面反射与光源的镜面反射成分相对应。
- 材料的环境光反射和漫反射定义材料的颜色，两者通常是相似的，镜面反射通常是白或灰。

8.1.2 光源属性

光源有许多属性，如颜色、位置、方向等。通常使用glLightfv()和glLightiv()定义光源属性。后续内容主要介绍这2个函数。

【函数原型】

- void glLightfv(GLenum light, GLenum pname, GLfloat *params);
- void glLightiv(GLenum light, GLenum pname, GLint *params);

【功能】设置光源参数。

【参数】

- light：指定待设置的光源，用形式为GL_LIGHTi的符号常数表示，$0 \leq i < 8$。
- pname：标识待设置的属性。
- params：给出pname对应的属性值。

下面按照光源的属性分类描述pname和params的取值。

1. 颜色

（1）环境光成分。

【调用形式】

- glLightfv(GLenum light, GL_ AMBIENT, float *params)
- glLightiv(GLenum light, GL_ AMBIENT, int *params)

【功能】用于指定环境光成分的RGBA强度。

【说明】默认值为(0, 0, 0, 1)，表示没有环境光。

（2）漫反射成分。

【调用形式】

- glLightfv(GLenum light, GL_ DIFFUSE, float *params)
- glLightiv(GLenum light, GL_ DIFFUSE, int *params)

【功能】用于指定漫反射成分的RGBA强度。

【说明】对于GL_LIGHT0，默认值为(1, 1, 1, 1)，对于其他光源，默认值为(0, 0, 0, 0)。

（3）镜面反射成分。

【调用形式】

- glLightfv(GLenum light, GL_ SPECULAR, float *params)
- glLightiv(GLenum light, GL_ SPECULAR, int *params)

【功能】用于指定镜面反射成分的RGBA强度。

【说明】对于GL_LIGHT0，默认值为(1, 1, 1, 1)，对于其他光源，默认值为(0, 0, 0, 0)。

2. 位置

【调用形式】

- glLightfv(GLenum light, GL_POSITION, float *params)
- glLightiv(GLenum light, GL_POSITION, int *params)

【功能】用于指定光源位置的齐次坐标 (x, y, z, w) 。

【说明】

- 若 $w \neq 0$ ，表示该光源是位置光源，距离场景较近，位置用齐次坐标 (x, y, z, w) 表示。
- 若 $w = 0$ ，表示该光源是方向光源，距离场景无限远，方向用从 (x, y, z) 到坐标原点的射线方向表示。
- (x, y, z, w) 的默认值为 $(0, 0, 1, 0)$ 。

下列语句将GL_LIGHT0定义为方向光源（需C99支持）。

```
glLightfv(GL_LIGHT0, GL_POSITION, (const float[]) { 1, 0, 1, 0});
```

3. 衰减

OpenGL使用 $1/\left(k_c + k_e d + k_q d^2\right)$ 衰减光线强度。其中，d 为光源位置到物体顶点之间的距离，k_c 为常数衰减因子（默认值为1），k_e 为线性衰减因子（默认值为0），k_q 为二次衰减因子（默认值为0）。这些衰减因子通常选用下列调用形式设置（见名知义，不再解释）。

- glLightf(GLenum light, GL_CONSTANT_ATTENUATION, float param)
- glLightf(GLenum light, GL_LINEAR_ATTENUATION, float param)
- glLightf(GLenum light, GL_QUADRATIC_ATTENUATION, float param)
- glLightfv(GLenum light, GL_CONSTANT_ATTENUATION, float *params)
- glLightfv(GLenum light, GL_LINEAR_ATTENUATION, float *params)
- glLightfv(GLenum light, GL_QUADRATIC_ATTENUATION, float *params)

【注】方向光源不需要随距离的增加而衰减光强度，只有位置光源才需要考虑光线强度的衰减。

4. 举例说明

下面语句定义了一个编号为GL_LIGHT0的光源（需C99支持）。

```
glLightfv(GL_LIGHT0, GL_AMBIENT, (const float[]) {0, 0, 0, 1});
// 环境光成分的RGBA强度
glLightfv(GL_LIGHT0, GL_DIFFUSE, (const float[]) {1, 1, 1, 1});
// 漫反射成分的RGBA强度
glLightfv(GL_LIGHT0, GL_SPECULAR, (const float[]) {1, 1, 1, 1});
// 镜面反射成分的RGBA强度
glLightfv(GL_LIGHT0, GL_POSITION, (const float[]) {1, 1, 1, 1});
// 光源位置（齐次坐标）
```

8.1.3 材质属性

物体表面材料有许多属性，如漫反射系数、环境光反射系数、镜面反射系数等。通常使用glMaterialfv()定义材质属性。后续内容主要介绍这个函数。

【函数原型】void glMaterialfv(GLenum face, GLenum pname, GLfloat *params);

【功能】为物体指定当前材质的某一属性。

【参数】

- face可以选用GL_FRONT、GL_BACK或GL_FRONT_AND_BACK。
- pname标识待设置的属性。
- params给出pname对应的属性值（数组）。

1. 漫反射和环境光反射

（1）漫反射系数。

【调用形式】glMaterialfv(GLenum face, GL_DIFFUSE, float *params)

【功能】用于定义物体对光源漫反射成分的反射系数。

【说明】漫反射系数对物体的颜色起最重要的作用，默认值为(0.8, 0.8, 0.8, 1)。

（2）环境光反射系数。

【调用形式】glMaterialfv(GLenum face, GL_AMBIENT, float *params)

【功能】用于定义物体对环境光的反射系数。

【说明】

- 默认值为(0.2, 0.2, 0.2, 1)。
- 环境光反射系数影响物体的整体颜色，因为直射到物体上的漫反射光最亮，而没有被直射的物体的环境光反射最明显。
- 物体的环境光反射强度受全局环境光和光源环境光成分的双重影响。

（3）同时指定漫反射系数和环境光反射系数。

在现实世界中，漫反射系数和环境光反射系数通常是相同的，因此OpenGL提供一种简便的方法为它们指定相同的值。

【调用形式】glMaterialfv(GLenum face, GL_AMBIENT_AND_DIFFUSE, float *params)

【功能】用于同时指定漫反射系数和环境光反射系数。

【说明】默认值为(0.8, 0.8, 0.8, 1)。

2. 镜面反射

（1）镜面反射系数。

【调用形式】glMaterialfv(GLenum face, GL_SPECULAR, float *params)

【功能】用于定义物体对光源镜面反射成分的反射系数，即亮斑的颜色。

【说明】默认值为(0, 0, 0, 1)。

（2）镜面反射指数。

【调用形式】

- glMaterialf(GLenum face, GL_SHINESS, float param)
- glMaterialfv(GLenum face, GL_SHINESS, float *params)

【功能】用于控制亮斑的大小和亮度，称为镜面反射指数或光洁度。

【说明】镜面反射指数的取值范围为[0, 128]，值越大，亮斑的尺寸越小且亮度越高。默认值为0。

3. 发光

【调用形式】glMaterialfv(GLenum face, GL_EMISSION, float *params)

【功能】用于定义发光体的发光强度，默认值为(0, 0, 0, 1)。

8.1.4 小结

表8-1对光源属性和材质属性设置的函数调用作了一个小结。

表8-1 光源属性和材质属性

属性	光源	材质
位置	glLightfv(light, GL_POSITION, xyzw)	
泛光	glLightfv(light, GL_AMBIENT, rgba)	glMaterialfv(face, GL_AMBIENT, rgba)
漫射	glLightfv(light, GL_DIFFUSE, rgba)	glMaterialfv(face, GL_DIFFUSE, rgba)
镜射	glLightfv(light, GL_SPECULAR, rgba)	glMaterialfv(face, GL_SPECULAR, rgba)
光洁度		glMaterialf(face, GL_SHINESS, s) glMaterialfv(face, GL_SHINESS, v)
发光		glMaterialfv(face, GL_EMISSION, rgba)

8.2 光照处理的几个例子

8.2.1 光照处理的相关函数调用

1. 光源和光照效果的启用和关闭

- glEnable(GL_LIGHTi)。打开第i号光源，$0 \leq i < 8$。
- glDisable(GL_LIGHTi)。关闭第i号光源，$0 \leq i < 8$。
- glEnable(GL_LIGHTING)。启用光照效果。
- glDisable(GL_LIGHTING)。禁止光照效果。

2. 明暗模式的选择

- glShadeModel(GL_SMOOTH)。光滑明暗模式，使用Gouraud算法。
- glShadeModel(GL_FLAT)。平面明暗模式，用其中一个顶点的颜色填充。

8.2.2 关于光照模型的例子

下列程序定义了一个光源和多个物体，演示了在光源相同、材质不同时的光照效果。图8-1给出了最终的显示效果。

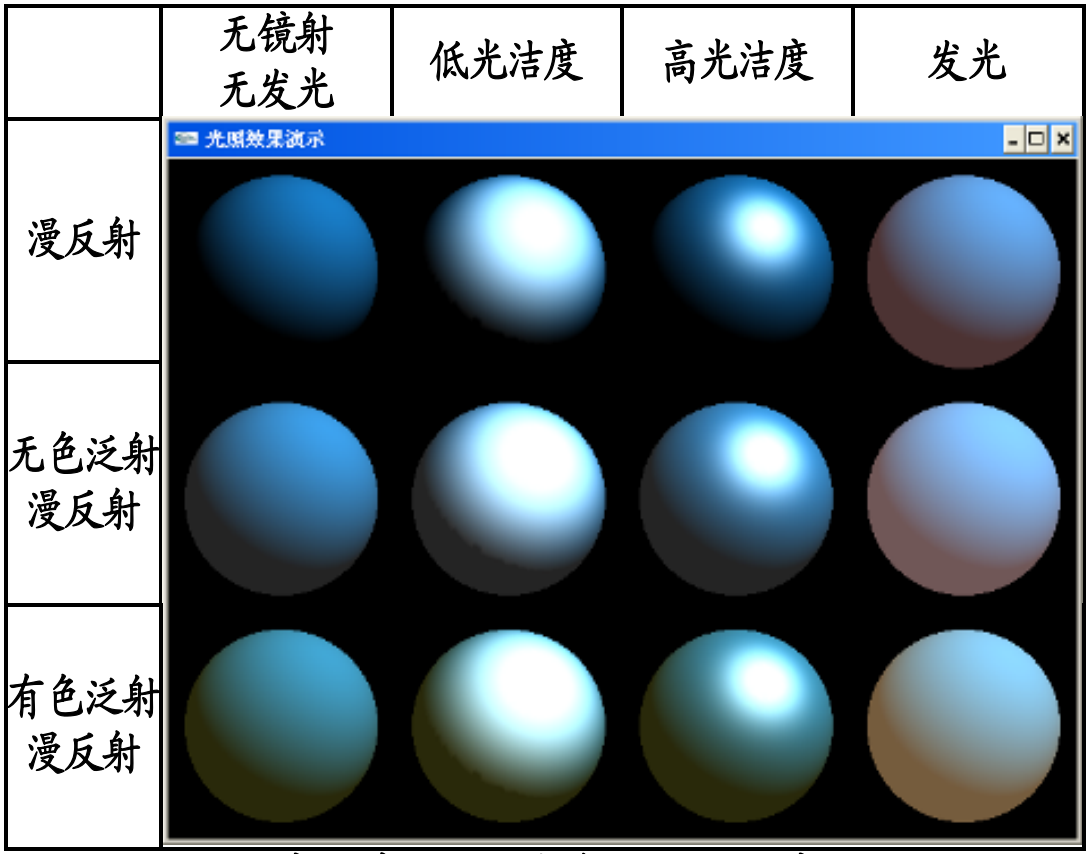


图8-1 光源相同、材质不同时的光照效果

```
// Material.c
#include<gl/freeglut.h>
void Viewport(int x, int y, int w, int h) // 创建视口，指定投影变换
{   glViewport(x, y, w, h); // 根据指定位置和大小创建视口
    glMatrixMode(GL_PROJECTION); // 投影矩阵栈
    glLoadIdentity(); // 当前矩阵改为单位矩阵，不再有其他投影变换
    gluPerspective(30, (float)w / h, 1, 1000); /* (2) 透视变换(视角30度，宽高比
w/h，近平面1，远平面1000) */
    glTranslatef(0, 0, -3.6); // (1) 场景适当远移
    glMatrixMode(GL_MODELVIEW); // 视图造型矩阵栈
    glLoadIdentity(); // 当前矩阵改为单位矩阵，不再有其他造型变换
}
void Light() // 光源特性
{   // 光源位置
    glLightfv(GL_LIGHT0, GL_POSITION, (const float[]) {2, 3, 3, 1});
    glEnable(GL_LIGHTING); // 激活光照
    glEnable(GL_LIGHT0); // 打开0号光源
}
```

```
void Material(float *mater[]) // 材质
{   int pname[] =
    {   GL_AMBIENT, // 环境光反射系数
        GL_DIFFUSE, // 漫反射系数
        GL_SPECULAR, // 镜面反射系数
        GL_SHININESS, // 光洁度(镜面反射指数)
        GL_EMISSION // 发光强度
    };
    for(int k = 0; k < 5; ++k) // 按顺序指定材质属性
        glMaterialfv(GL_FRONT, pname[k], mater[k]);
}
```

```
void Paint() // 场景绘制函数
{   // 建立材质数据库
    float no_mat[] = {0, 0, 0, 1}; // 无泛射, 漫反射或镜面反射
    float amb[] = {0.7, 0.7, 0.7, 1}; // 不带颜色的泛射
    float amb_color[] = {0.8, 0.8, 0.2, 1}; // 带颜色的泛射
    float diff[] = {0.1, 0.5, 0.8, 1}; // 漫反射
    float spec[] = {1, 1, 1, 1}; // 镜面反射
    float no_shin[] = {0}; // 光洁度(镜面反射指数)
    float low_shin[] = {4}; // 低光洁度
    float high_shin[] = {16}; // 高光洁度
    float emiss[] = {0.3, 0.2, 0.2, 0}; // 发光强度
```

```
float *mater[3][4][5] =
{   // 每行分别对应一个物体的材质属性
    // 每列分别对应泛射，漫反射，镜面反射，光洁度，发光
    no_mat, diff, no_mat, no_shin, no_mat, // 漫
    no_mat, diff, spec, low_shin, no_mat, // 漫，低
    no_mat, diff, spec, high_shin, no_mat, // 漫，高
    no_mat, diff, no_mat, no_shin, emiss, // 漫，发
    amb, diff, no_mat, no_shin, no_mat, // 泛，漫
    amb, diff, spec, low_shin, no_mat, // 泛，漫，低
    amb, diff, spec, high_shin, no_mat, // 泛，漫，高
    amb, diff, no_mat, no_shin, emiss, // 泛，漫，发
    amb_color, diff, no_mat, no_shin, no_mat, // 色，漫
    amb_color, diff, spec, low_shin, no_mat, // 色，漫，低
    amb_color, diff, spec, high_shin, no_mat, // 色，漫，高
    amb_color, diff, no_mat, no_shin, emiss // 色，漫，发
};
```

```
    int w = glutGet(GLUT_WINDOW_WIDTH) / 4; // 视口宽度
    int h = glutGet(GLUT_WINDOW_HEIGHT) / 3; // 视口高度
    glClear(GL_COLOR_BUFFER_BIT); // 清除颜色缓存
    glClear(GL_DEPTH_BUFFER_BIT); // 清除深度缓存
    // 按照3行4列绘制12个球
    for(int i = 0; i < 3; ++i)
        for(int j = 0; j < 4; ++j)
        {   Viewport(w * j, (2 - i) * h, w, h); // 第i行第j列
            Material(mater[i][j]); // 指定材质属性
            glutSolidSphere(0.8, 120, 60); // 实体球(半径, 经线数, 纬线数)
        }
    glFlush();
}
```

```
int main(int argc, char *argv[])
{   glutInit(&argc, argv);
    glutInitWindowSize(640, 480); // 窗口大小(4 : 3)
    glutCreateWindow("光照效果演示");
    glEnable(GL_DEPTH_TEST); // 打开深度测试, 用于比较远近
    Light(); // 光源特性
    glutDisplayFunc(Paint);
    glutMainLoop();
}
```

8.2.3 简单日地月系统的真实感绘制※

下列程序演示了日地月的公转和自转，地球上的白天黑夜和四季变化，月球的圆缺等效果。为了增强演示效果，还绘制了日地月的经纬线，并在赤道上放置了一个小球。运行时某一时刻的画面如图8-2所示。

在Paint()中，首先定义太阳，然后规定地球的公转角度和放置位置，考虑地球的黄赤角以保证地轴方向，规定自转角度并定义地球，最后规定月球的公转角度（自转角度不需要指定）和放置位置，并定义月球。

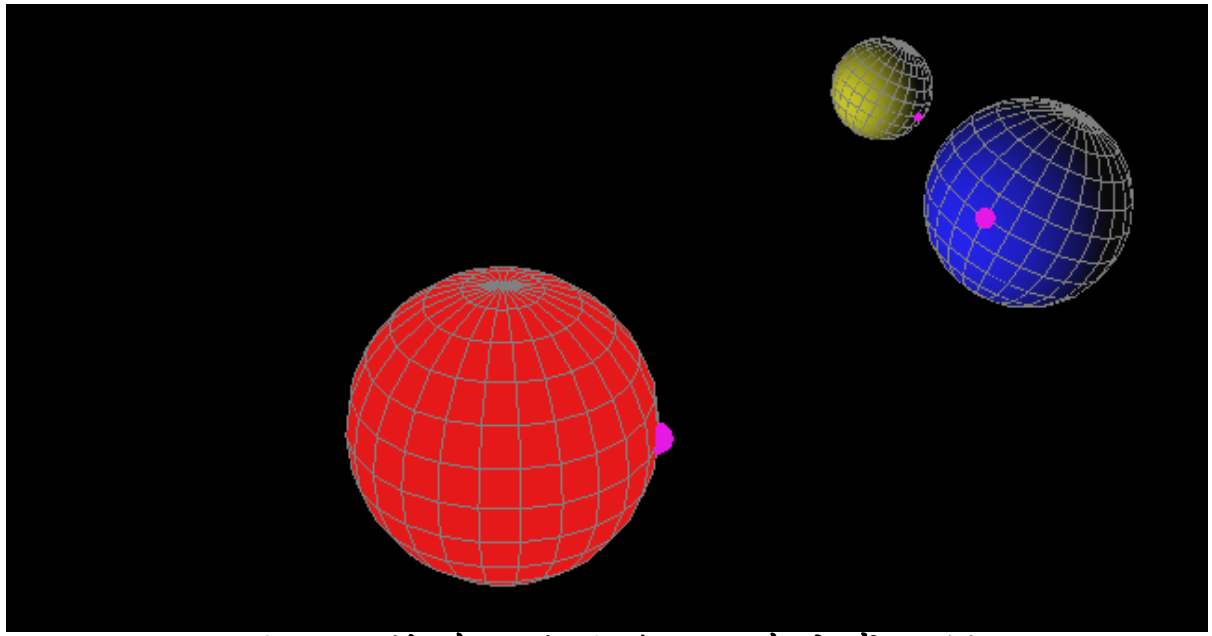


图8-2 简单日地月系统的真实感绘制

（1）预处理与共享变量。

```
// Planetary.c
#include<math.h> // fmod()等
#include<gl/freeglut.h>
```

```
float day = 0; // 地球自转角度
float month = 0; // 月亮公转角度(或自转角度, 两者相同)
float year = 0; // 地球公转角度
float sun = 0; // 太阳自转角度
```

（2）按键与窗口变化的响应。

```
void Keyboard(unsigned char key, int x, int y) // 键盘按键响应函数
{   if(key == 27) exit(0); // 按下Esc键退出
}
```

```
void Reshape(int w, int h) // 窗口变化响应函数
{   glViewport(0, 0, w, h); // 视口的位置和大小
    glMatrixMode(GL_PROJECTION); // 当前变换类型为投影变换
    glLoadIdentity(); // 当前矩阵为单位矩阵，不再有其他投影变换
    gluPerspective(30, (float)w / h, 1, 1000); /* 透视变换(y向张角，宽高比，近平面，远平面) */
    glTranslatef(0, 0, -8); // 适当远移
    glRotatef(30, 1, 0, 0); // 调整观察角度
    glMatrixMode(GL_MODELVIEW); // 当前变换类型为视图造型变换
    glLoadIdentity(); // 当前矩阵为单位矩阵，不再有其他造型变换
}
```

（3）日地月定义。

```
void Material(float r, float g, float b) // 使用颜色指定物体表面材质
{   glEnable(GL_LIGHTING); // 使用光照效果
    // 反射材质
    glMaterialfv(GL_FRONT, GL_DIFFUSE, (float[]) {r, g, b, 1});
}
```

```
void Color(float r, float g, float b) // 直接指定物体表面颜色
{   glDisable(GL_LIGHTING); // 不使用光照效果
    glColor3f(r, g, b); // 直接指定颜色
}
```

```
void Wire(float r) // 经纬线
{   Color(0.5, 0.5, 0.5); // 经纬线是灰色的
    glutWireSphere(r * 1.005, 24, 12); // 经纬线将星球包裹在内
}
```

```
void Ball(float r) // 在赤道上放置一个小球验证自转
{   glPushMatrix(); // 保存当前矩阵
    glTranslatef(-r, 0, 0); // 在赤道与负x轴交点处放置一个小球
    Color(0.9, 0.1, 0.9); // 小球是紫色的
    glutSolidSphere(r * 0.08, 12, 6); // 小球
    glPopMatrix(); // 恢复当前矩阵, 避免上述变换影响到其他物体
}
```

```
void Sun(float r) // 太阳
{   glPushMatrix(); // 保存当前矩阵
    glRotatef(90, -1, 0, 0); // 将物体的高度方向从z调整到y
    Color(0.9, 0.1, 0.1); // 太阳是红色的
    glutSolidSphere(r, 24, 12); // 太阳
    Wire(r); // 经纬线
    Ball(r); // 在赤道上放置一个小球验证自转
    glPopMatrix(); // 恢复当前矩阵, 避免上述变换影响到其他物体
}
```

```
void Planet(float r) // 地球
{   glPushMatrix(); // 保存当前矩阵
    glRotatef(90, -1, 0, 0); // 将物体的高度方向从z调整到y
    Material(0.1, 0.1, 0.9); // 地球是蓝色的
    glutSolidSphere(r, 24, 12); // 地球
    Wire(r); // 经纬线
    Ball(r); // 在赤道上放置一个小球验证自转
    glPopMatrix(); // 恢复当前矩阵, 避免上述变换影响到其他物体
}
```

```
void Moon(float r) // 月亮
{   glPushMatrix(); // 保存当前矩阵
    glRotatef(90, -1, 0, 0); // 将物体的高度方向从z调整到y
    Material(0.75, 0.75, 0.1); // 月亮是浅黄色的
    glutSolidSphere(r, 24, 12); // 月亮
    Wire(r); // 经纬线
    Ball(r); // 在赤道上放置一个小球验证自转
    glPopMatrix(); // 恢复当前矩阵, 避免上述变换影响到其他物体
}
```

（4）场景绘制。

```
void Paint() // 场景绘制函数
{   glClear(GL_COLOR_BUFFER_BIT); // 清除颜色缓存
    glClear(GL_DEPTH_BUFFER_BIT); // 清除深度缓存
    glLoadIdentity(); // 不再有其他造型变换
    // 太阳: 自转
    {   glPushMatrix();
        glRotatef(sun, 0, 1, 0); // 太阳自转
        Sun(0.4); // 太阳
        glPopMatrix(); // 取消太阳自转的影响
    }
    // 地球: 自转-->黄赤角-->保证地轴方向-->位置-->公转
    glRotatef(year, 0, 1, 0); // (5) 地球公转
    // 指定光源
    {   // 指定光源方向, 使光源对着地球
        glLightfv(GL_LIGHT0, GL_POSITION, (float[]) { -1, 0, 0, 0});
        // 指定光源的漫反射成分
        glLightfv(GL_LIGHT0, GL_DIFFUSE, (float[]) {1, 1, 1, 1});
        glEnable(GL_LIGHT0); // 启用光源
    }
```

```
    glTranslatef(2, 0, 0); // (4) 地球相对太阳的初始位置
    glRotatef(-year, 0, 1, 0); // (3) 取消公转的影响, 保证地轴方向不变
    glRotatef(-23.5, 0, 0, 1); // (2) 黄赤交角, 公转平面与自转平面的夹角
    {   glPushMatrix();
        glRotatef(day, 0, 1, 0); // (1) 地球自转
        Planet(0.3); // 地球
        glPopMatrix(); // 取消地球自转的影响
    }
    // 月亮: 自转-->位置-->公转-->地球黄赤角--> ...
    glRotatef(month, 0, 1, 0); // (2) 月亮公转
    glTranslatef(0.6, 0, 0); // (1) 月亮相对地球的初始位置
    // 月亮自转周期与公转周期相同, 不需要指定
    Moon(0.15); // 月亮
    glutSwapBuffers(); // 交换颜色缓存
}
```

（5）场景刷新。

```
void Timer(int millis) // 定时器响应函数(场景刷新)
{   float base = 36; // 基准的角度增量(对应0.1天/帧)
    float dday = base * 0.1; // 地球自转角度的增量(较慢, 0.1倍正常速度)
    float dmonth = base / 27.3217; // 月亮公转角度的增量(恒星月)
    float dyear = base / 365.2564; // 地球公转角度的增量(恒星年)
    float dsun = base / 25.38; // 太阳自转角度的增量(恒星周期)
    day = fmod(day + dday, 360); // 计算地球自转角度(较慢)
    month = fmod(month + dmonth, 360); // 计算月亮公转角度
    year = fmod(year + dyear, 360); // 计算地球公转角度
    sun = fmod(sun + dsun, 360); // 计算太阳自转角度
    glutPostRedisplay(); // 调用场景绘制函数
    glutTimerFunc(millis, Timer, millis); /* 指定定时器函数(间隔毫秒数, 函数
名, 函数参数值) */
}
```

（6）窗口初始化与回调函数注册。

```
int main(int argc, char *argv[])
{   glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE); // 双缓存
    glutCreateWindow("一个简单的日地月系统");
    // 当前窗口初始化
    glutFullScreen(); // 使用全屏幕窗口
    glEnable(GL_DEPTH_TEST); // 打开深度测试, 用于比较远近
    // 注册回调函数
    glutDisplayFunc(Paint); // 指定场景绘制函数
    glutReshapeFunc(Reshape); // 指定窗口变化响应函数
    glutKeyboardFunc(Keyboard); // 指定键盘按键响应函数
    glutTimerFunc(30, Timer, 30); /* 指定定时器函数(间隔毫秒数, 函数名,
函数参数值) */
    glutMainLoop();
}
```

8.3 融合

8.3.1 融合的含义

融合处理常用于透明或半透明的物体。在RGBA模式下，假设源色（新引入）为 (R_s, G_s, B_s, A_s)，目标色（存在帧缓存中）为 (R_d, G_d, B_d, A_d)，源因子为 (S_r, S_g, S_b, S_a)，目标因子为 (D_r, D_g, D_b, D_a)，则融合效果为

$$(R_sS_r + R_dD_r,\ G_sS_g + G_dD_g,\ B_sS_b + B_dD_b,\ A_sS_a + A_dD_a)$$

然后再归一。这里的关键是如何设定融合因子 (S_r, S_g, S_b, S_a) 和 (D_r, D_g, D_b, D_a) 来实现不同的融合效果。

$$I = (1 - k_t) I_{\text{refl}} + k_t I_{\text{trans}}$$

$$I = I_s S + I_d D$$

源 新引入　　目的 缓存

RGBA，A不透明程度

8.3.2 相关函数

1. 融合函数

OpenGL只能通过融合函数glBlendFunc()设定融合因子。

【函数原型】void glBlendFunc(GLenum sfactor, GLenum dfactor);

【功能】选择融合因子。

【参数】

- sfactor是源因子，可以选用表8-2中的所有值，最常用的是GL_SRC_ALPHA。
- dfactor是目标因子，可以选用表8-2中的前8个值，最常用的是GL_ONE或GL_ONE_MINUS_SRC_ALPHA。

【说明】

- glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)常用于实现透明效果。
- glBlendFunc(GL_SRC_ALPHA, GL_ONE)常用于优化反走样效果。

$$I=(1-k_t)I_{\text{refl}}+k_tI_{\text{trans}}$$

$$I = I_sS + I_dD$$

源 目的

新引入 缓存

RGBA，A不透明程度

表8-2　源因子和目标因子的可能值及含义

可能值	含义
GL_ZERO	$(0,0,0,0)$
GL_ONE	$(1,1,1,1)$
GL_DST_COLOR	(R_d, G_d, B_d, A_d)
GL_ONE_MINUS_DST_COLOR	$(1-R_d, 1-G_d, 1-B_d, 1-A_d)$
GL_SRC_ALPHA	(A_s, A_s, A_s, A_s)
GL_ONE_MINUS_SRC_ALPHA	$(1-A_s, 1-A_s, 1-A_s, 1-A_s)$
GL_DST_ALPHA	(A_d, A_d, A_d, A_d)
GL_ONE_MINUS_DST_ALPHA	$(1-A_d, 1-A_d, 1-A_d, 1-A_d)$
GL_SRC_ALPHA_SATURATE	$(f,f,f,f),\ f=\min(A_s, 1-A_d)$

$$I=(1-k_t)I_{\text{refl}}+k_tI_{\text{trans}}$$

$$I = I_sS + I_dD$$

源　　目的

新引入　　缓存

RGBA，A不透明程度

2. 融合的启用和关闭

- glEnable(GL_BLEND)。启用融合处理。
- glDisable(GL_BLEND)。关闭融合处理。

8.3.3 举例

下列程序首先绘制一个半透明的红色矩形，不透明程度为0.7；其次绘制一个完全不透明的绿色矩形；最后绘制一个透明的蓝色矩形，不透明程度为0.3。运行结果如图8-3所示。

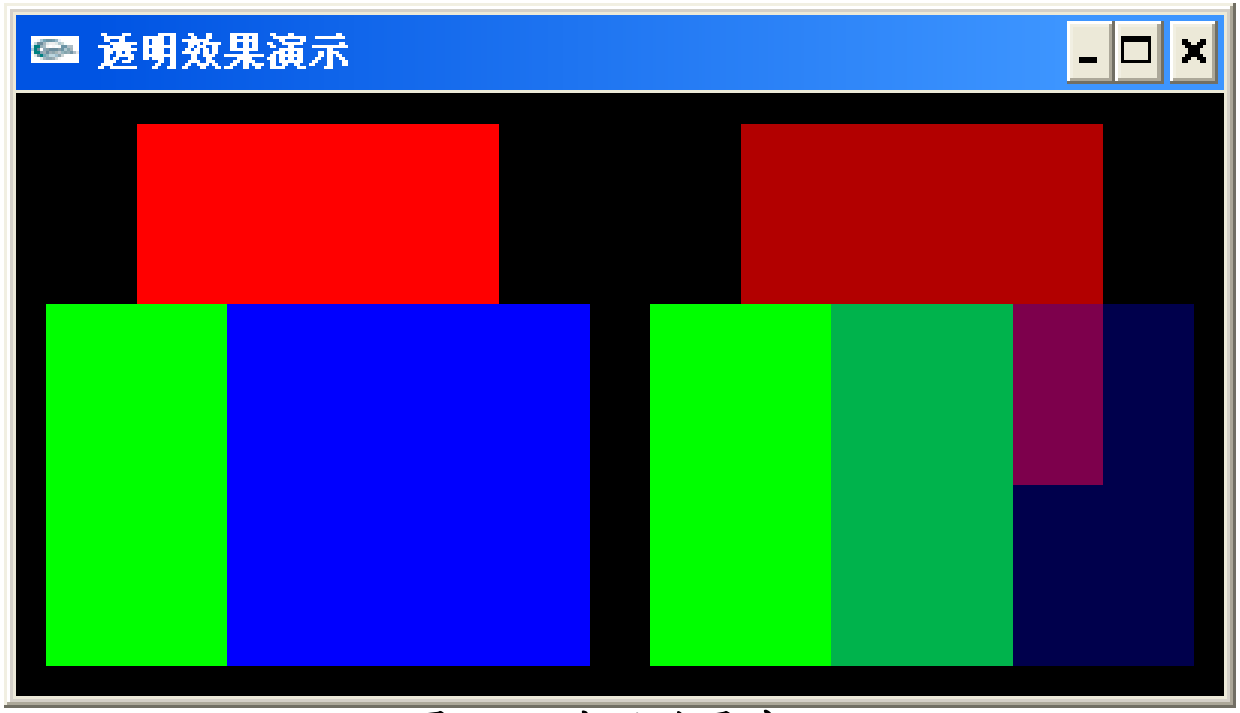


图8-3 透明效果演示

```
// Blend.c
#include<gl/freeglut.h>
void Rects() // 三个方块
{   // 三个不同颜色和透明度的方块重合，颜色融合的效果
    glLoadIdentity(); // 红色方块在默认位置，不再有其他造型变换
    glColor4f(1, 0, 0, 0.7); // 红色，半透明
    glRectf(-0.6, -0.3, 0.6, 0.9);
    glTranslatef(0, 0, 0.2); // 绿色方块移近0.2
    glColor4f(0, 1, 0, 1); // 绿色，不透明
    glRectf(-0.9, -0.9, 0.3, 0.3);
    glTranslatef(0, 0, 0.2); // 蓝色方块再移近0.2
    glColor4f(0, 0, 1, 0.3); // 蓝色，透明
    glRectf(-0.3, -0.9, 0.9, 0.3);
}
void Paint() // 场景绘制函数
{   int w = glutGet(GLUT_WINDOW_WIDTH) / 2; // 视口宽度
    int h = glutGet(GLUT_WINDOW_HEIGHT); // 视口高度
    glClear(GL_COLOR_BUFFER_BIT); // 清除颜色缓存
```

```
    glViewport(0, 0, w, h); // 左侧视口
    glDisable(GL_BLEND); // 禁用融合
    Rects(); // 三个方块
    glViewport(w, 0, w, h); // 右侧视口
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    // 设置融合因子
    glEnable(GL_BLEND); // 启用融合
    Rects();
    glFlush();
}
int main(int argc, char *argv[])
{   glutInit(&argc, argv);
    glutInitWindowSize(400, 200); // 窗口大小(2 : 1)
    glutCreateWindow("透明效果演示");
    glutDisplayFunc(Paint);
    glutMainLoop();
}
```

8.4 反走样※

由于计算机以离散点生成图形，生成的图形必然与真实景物存在差距，这种差距表现为：直线或光滑曲面的锯齿、花纹失去原有色彩形状、细小物体在画面中消失等，这些都叫做走样。

反走样可以减少这种情况。反走样就是把原来边界的锯齿部分用低饱和度的点补上，这样既不影响整体轮廓，又获得较好的平滑效果。

8.4.1 相关函数调用

1. 启用反走样

- glEnable(GL_POINT_SMOOTH)。启用点的反走样。
- glEnable(GL_LINE_SMOOTH)。启用线段的反走样。
- glEnable(GL_POLYGON_SMOOTH)。启用多边形的反走样。

2. 启用融合

- glEnable(GL_BLEND)。启用融合。
- glBlendFunc(sfactor, dfactor)。选择融合因子。

【说明】sfactor通常选用GL_SRC_ALPHA，dfactor通常选用GL_ONE或GL_ONE_MINUS_SRC_ALPHA。

8.4.2 举例说明

下列程序分别使用顶点、边框和填充方式绘制1个顺时针旋转了15度的正方形，用于演示点、线段和多边形的反走样效果。运行结果如图8-4所示。

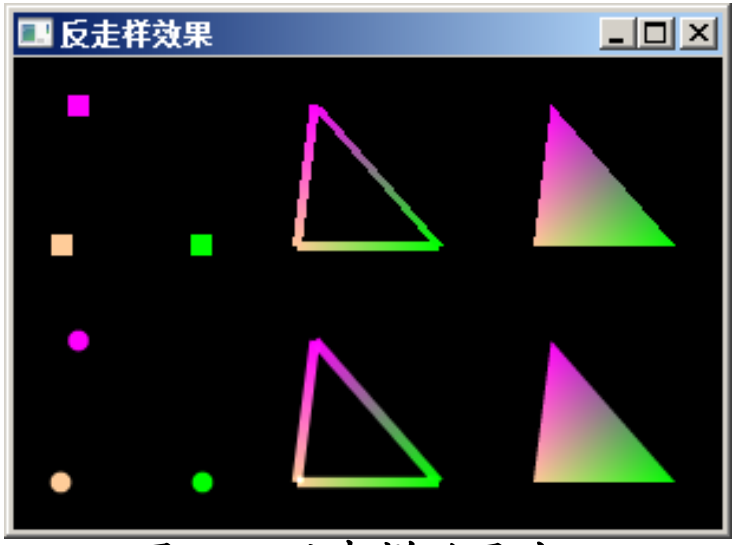
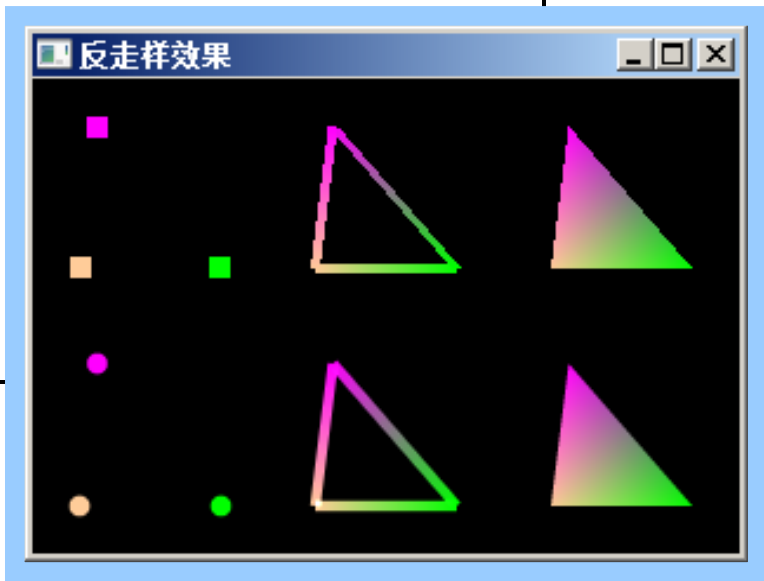


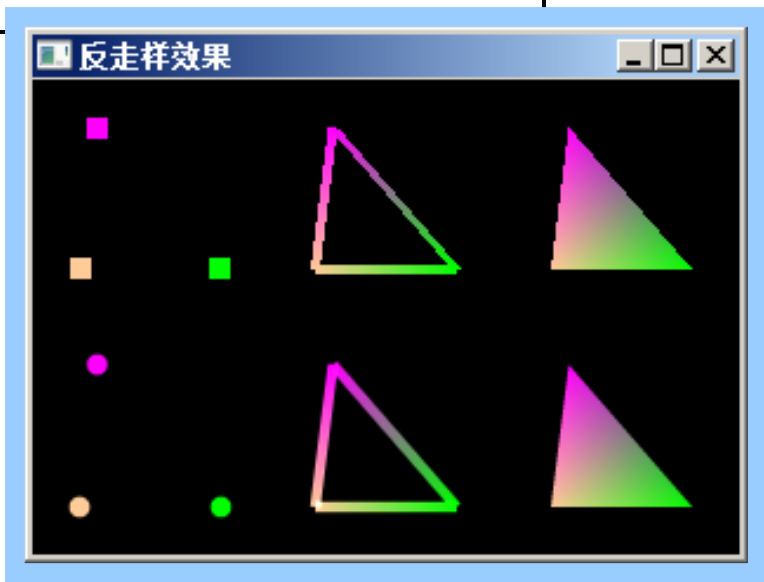
图8-4 反走样效果演示

```
// Smooth.c
#include<GL/freeglut.h>
void Hint() // 当前窗口初始化(点宽, 线宽和融合效果)
{   glPointSize(8.8); // 点宽=8.8
    glLineWidth(4.4); // 线宽=4.4
    glBlendFunc(GL_SRC_ALPHA, GL_ONE); // 融合效果
}
void Triangle() // 定义彩色三角形
{   glBegin(GL_TRIANGLES); // 定义三角形(逆时针指定顶点)
    {   glColor3f(1, 0.8, 0.6); // 茶色
        glVertex2f(-0.6, -0.6); // 左下顶点
        glColor3f(0, 1, 0); // 绿色
        glVertex2f(0.6, -0.6); // 右下顶点
        glColor3f(1, 0, 1); // 紫色
        glVertex2f(-0.45, 0.6); // 上方顶点
    }
    glEnd();
}
```

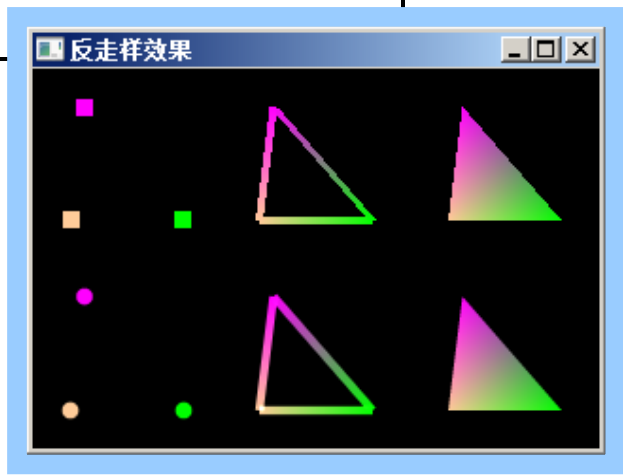


```
void EnableSmooth() // 启用反走样
{   glEnable(GL_POINT_SMOOTH); // 启用点的反走样
    glEnable(GL_LINE_SMOOTH); // 启用线段反走样
    glEnable(GL_POLYGON_SMOOTH); // 启用多边形反走样
    glEnable(GL_BLEND); // 启用融合
}
```

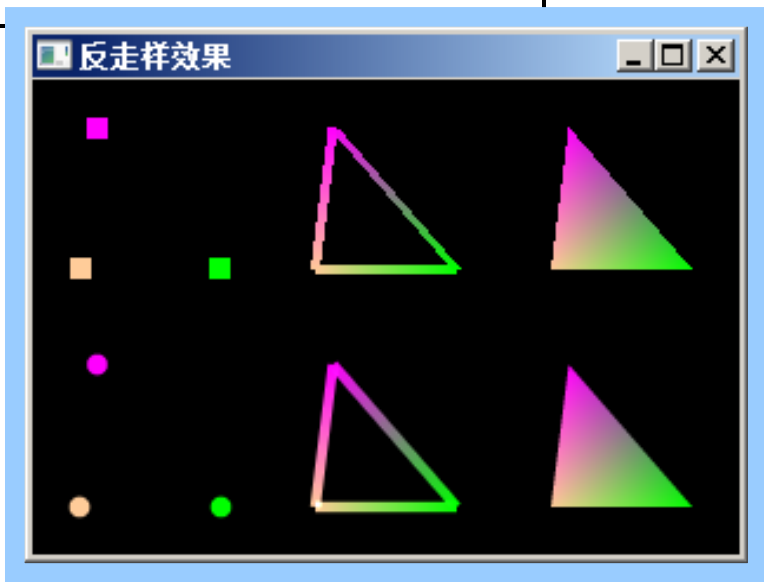
```
void DisableSmooth() // 取消反走样
{   glDisable(GL_POINT_SMOOTH); // 取消点的反走样
    glDisable(GL_LINE_SMOOTH); // 取消线段反走样
    glDisable(GL_POLYGON_SMOOTH); // 取消多边形反走样
    glDisable(GL_BLEND); // 取消融合
}
```



```
void Paint() // 场景绘制函数
{   int w = glutGet(GLUT_WINDOW_WIDTH) / 3; // 视口宽度
    int h = glutGet(GLUT_WINDOW_HEIGHT) / 2; // 视口高度
    glClear(GL_COLOR_BUFFER_BIT); // 清除颜色缓冲区
    EnableSmooth(); // 启用反走样，下行
    glViewport(0, 0, w, h); // 第0行第0列
    glPolygonMode(GL_FRONT, GL_POINT); // 正面，只绘制顶点
    Triangle();
    glViewport(w, 0, w, h); // 第0行第1列
    glPolygonMode(GL_FRONT, GL_LINE); // 正面，只绘制边框
    Triangle();
    glViewport(2 * w, 0, w, h); // 第0行第2列
    glPolygonMode(GL_FRONT, GL_FILL); // 正面，绘制填充多边形
    Triangle();
```



```
    DisableSmooth(); // 取消反走样，上行
    glViewport(0, h, w, h); // 第1行第0列
    glPolygonMode(GL_FRONT, GL_POINT); // 正面，只绘制顶点
    Triangle();
    glViewport(w, h, w, h); // 第1行第1列
    glPolygonMode(GL_FRONT, GL_LINE); // 正面，只绘制边框
    Triangle();
    glViewport(2 * w, h, w, h); // 第1行第2列
    glPolygonMode(GL_FRONT, GL_FILL); // 正面，绘制填充多边形
    Triangle();
    glFlush(); // 强制OpenGL命令序列在有限时间内完成执行
}
```



```
int main(int argc, char *argv[])
{   glutInit(&argc, argv); // 初始化GLUT，记录main()的参数
    glutInitWindowSize(300, 200); // 程序窗口的初始大小(3 : 2)
    glutCreateWindow("反走样效果"); // 创建程序窗口，指定窗口标题
    Hint(); // 当前窗口初始化(点宽，线宽和融合效果)
    glutDisplayFunc(Paint); // 指定场景绘制函数
    glutMainLoop(); // 开始循环执行OpenGL命令
}
```

8.5# 纹理映射

纹理映射是把图像应用到物体表面的一种技术，类似于在物体表面贴上印花图案。在OpenGL中，纹理图像在纹理空间中创建，采用 (s,t) 坐标系统，坐标范围通常为0~1。纹理通常是一维或二维图像。

在OpenGL中使用纹理映射的基本步骤如下。

（1）定义纹理，即指定纹理图像。

（2）控制滤波。

（3）给出顶点的纹理坐标和几何坐标。

【注】纹理映射只能在RGBA模式下使用，不适用于颜色索引模式。

8.5.1 纹理定义

1. 二维纹理定义函数

【函数原型】glTexImage2D(GLenum target, int level, int components, int width, int height, int border, GLenum format, GLenum type, void *pixels)

【功能】用于指定一个二维纹理图像。

【参数】

- target：只能使用GL_TEXTURE_2D，使用其他值会产生错误。
- level：多分辨率纹理图像的级数，通常为0（只有一种分辨率）。
- components：纹理图像中的颜色分量数，可选1、2、3或4。
- width和height：纹理图像的宽度和高度，必须为2^n+2×border。
- border：边界的宽度，必须为0或1。
- format：像素数据的格式，可用GL_RED、GL_GREEN、GL_BLUE、GL_ALPHA、GL_RGB、GL_RGBA、GL_BGR_EXT、GL_BGRA_EXT等。
- type：像素数据的类型，可选GL_BYTE、GL_UNSIGNED_BYTE、GL_SHORT、GL_UNSIGNED_SHORT、GL_INT、GL_UNSIGNED_INT、GL_FLOAT等。
- pixels：纹理图像数据（含边界）。

2. 一维纹理定义函数

【调用形式】glTexImage1D(GLenum target, int level, int components, int width, int border, GLenum format, GLenum type, void *pixels)

【功能】用于指定一个一维纹理图像。

【参数】

- target：只能使用GL_TEXTURE_1D，使用其他值会产生错误。
- level：多分辨率纹理图像的级数，通常为0（只有一种分辨率）。
- components：纹理图像中的颜色分量数，可选1、2、3或4。
- width：纹理图像的宽度，必须为2^n+2×border。
- border：边界的宽度，必须为0或1。
- format：像素数据的格式，可用GL_RED、GL_GREEN、GL_BLUE、GL_ALPHA、GL_RGB、GL_RGBA、GL_BGR_EXT、GL_BGRA_EXT等。
- type：像素数据的类型，可选GL_BYTE、GL_UNSIGNED_BYTE、GL_SHORT、GL_UNSIGNED_SHORT、GL_INT、GL_UNSIGNED_INT、GL_FLOAT等。
- pixels：纹理图像数据（含边界）。

8.5.2 控制滤波

这里以二维纹理为例说明。原始的纹理图像是一个方形图像，要把它映射到奇形怪状的物体表面上，一个纹素对应一个屏幕像素一般是很难做到的，需要对纹理进行放大或缩小。

1. 相关函数

【函数原型】void glTexParameteri(GLenum target, GLenum pname, GLint param);

【功能】设置纹理参数。

【参数】

- target表示目标纹理，只能选用GL_TEXTURE_1D或GL_TEXTURE_2D。
- pname指定待设置纹理参数的名称，这里只介绍用于纹理放大与纹理缩小的GL_TEXTURE_MAG_FILTER和GL_TEXTURE_MIN_FILTER。
- param指定pname指定的纹理参数的参数值，这里只介绍用于纹理放大与纹理缩小的GL_NEAREST和GL_LINEAR。

```
void glTexParameteri(
    GLenum target,
    GLenum pname,
    GLint param);
```

2. 纹理放大与纹理缩小

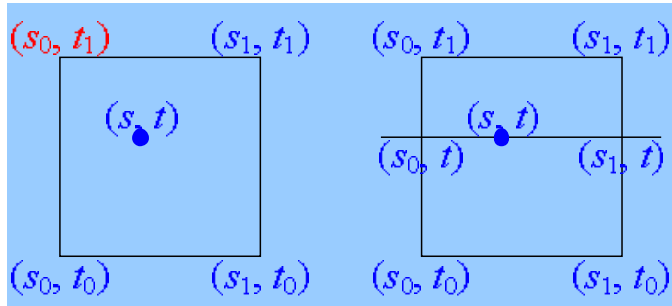
pname选用GL_TEXTURE_MAG_FILTER表示通过param指定使用哪种纹理放大函数，意思是当把应用纹理的像素映射到小于或等于一个纹素的区域上时使用哪种纹理映射方法。

pname选用GL_TEXTURE_MIN_FILTER表示通过param指定使用哪种纹理缩小函数，意思是当把应用纹理的像素映射到大于一个纹素的区域上时使用哪种纹理映射方法。

3. 纹理映射方法

param选用GL_NEAREST表示使用最靠近像素（该像素使用纹理）中心的纹素值与像素值结合。

param选用GL_LINEAR表示使用最靠近像素（该像素使用纹理）中心的四个纹素值的加权平均与像素值结合。



8.5.3 纹理坐标

1. 纹理坐标的设置

纹理图像是方形的，纹理坐标可定义成齐次坐标(S, T, R, Q)。用

void glTexCoord{1234}{sifd}[v](TYPE coords)

设置当前纹理坐标，此后调用glVertex*()所产生的顶点都赋予当前的纹理坐标。

```
void glTexCoord1d(GLdouble s);                                  void glTexCoord1dv(const GLdouble *v);
void glTexCoord1f(GLfloat s);                                   void glTexCoord1fv(const GLfloat *v);
void glTexCoord1i(GLint s);                                     void glTexCoord1iv(const GLint *v);
void glTexCoord1s(GLshort s);                                   void glTexCoord1sv(const GLshort *v);
void glTexCoord2d(GLdouble s, GLdouble t);                      void glTexCoord2dv(const GLdouble *v);
void glTexCoord2f(GLfloat s, GLfloat t);                        void glTexCoord2fv(const GLfloat *v);
void glTexCoord2i(GLint s, GLint t);                            void glTexCoord2iv(const GLint *v);
void glTexCoord2s(GLshort s, GLshort t);                        void glTexCoord2sv(const GLshort *v);
void glTexCoord3d(GLdouble s, GLdouble t, GLdouble r);          void glTexCoord3dv(const GLdouble *v);
void glTexCoord3f(GLfloat s, GLfloat t, GLfloat r);             void glTexCoord3fv(const GLfloat *v);
void glTexCoord3i(GLint s, GLint t, GLint r);                   void glTexCoord3iv(const GLint *v);
void glTexCoord3s(GLshort s, GLshort t, GLshort r);             void glTexCoord3sv(const GLshort *v);
void glTexCoord4d(GLdouble s, GLdouble t, GLdouble r, GLdouble q);  void glTexCoord4dv(const GLdouble *v);
void glTexCoord4f(GLfloat s, GLfloat t, GLfloat r, GLfloat q);  void glTexCoord4fv(const GLfloat *v);
void glTexCoord4i(GLint s, GLint t, GLint r, GLint q);          void glTexCoord4iv(const GLint *v);
void glTexCoord4s(GLshort s, GLshort t, GLshort r, GLshort q);  void glTexCoord4sv(const GLshort *v);
```

2. 自动生成纹理坐标

有时不需要或者不方便为每个物体顶点赋予纹理坐标，可以使用glTexGen*()自动产生纹理坐标。

（1）选取纹理坐标生成函数。

【函数原型】void glTexGeni(GLenum coord, GLenum pname, GLint param);

【参数】

- coord指明哪个分量自动产生，可选GL_S、GL_T、GL_R和GL_Q。
- pname必须使用GL_TEXTURE_GEN_MODE。
- param用于指定纹理坐标生成函数的符号常量名，通常选用GL_OBJECT_LINEAR（在物空间中计算）或GL_EYE_LINEAR（在像空间中计算）。

（2）指定生成函数的参数。

【函数原型】

- void glTexGendv(GLenum coord, GLenum pname, GLdouble *params);
- void glTexGenfv(GLenum coord, GLenum pname, GLfloat *params);
- void glTexGeniv(GLenum coord, GLenum pname, GLint *params);

【参数】

- coord指明哪个分量自动产生，可选GL_S、GL_T、GL_R和GL_Q。
- pname用于指定纹理坐标生成函数的符号常量名，只能选用GL_OBJECT_PLANE（在物空间中计算）或GL_EYE_PLANE（在像空间中计算）。
- params指定生成函数 $g = ax + by + cz + dw$ 的系数 (a,b,c,d) 。

【说明】参数pname的值必须和glTexGeni()等函数中的param参数值对应。

8.5.4 举例说明

1. 指定纹理坐标的例子

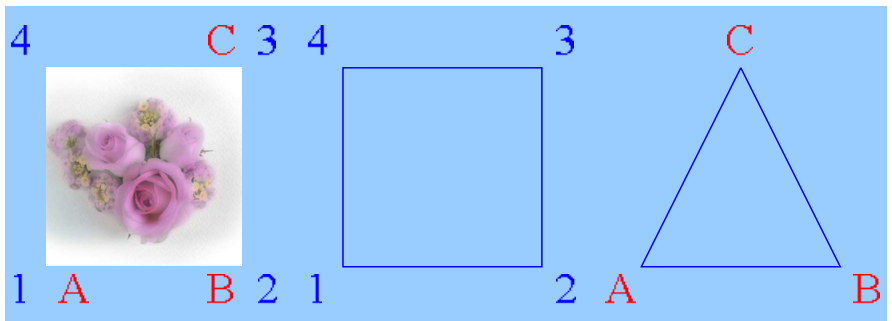
下列程序使用图像文件定义纹理图像（使用256×256的24位位图文件Flower.bmp），并将纹理图像映射到一个灰色矩形和一个灰色三角形上，便于看出纹理映射的特点。程序运行结果如图8-5所示。可以看出，最终屏幕像素的颜色是物体颜色和纹理像素颜色的结合，并且该程序将纹理图像映射到三角形时产生了变形。

图8-5 二维纹理映射示例

```
// Texture2d.c
#include<stdio.h> // 文件读写
#include<windows.h> // 位图文件相关数据结构, 可抄出
#include<gl/freeglut.h>
void Reshape(int w, int h) // 窗口变化响应函数
{   glViewport(0, 0, w, h); // 根据指定位置和大小创建视口
    glMatrixMode(GL_PROJECTION); // 投影矩阵栈
    glLoadIdentity(); // 当前矩阵改为单位矩阵, 不再有其他投影变换
    gluOrtho2D(0, 2, 0, 1); // 正投影(左, 右, 下, 上)
    glMatrixMode(GL_MODELVIEW); // 视图造型矩阵栈
    glLoadIdentity(); // 当前矩阵改为单位矩阵, 不再有其他造型变换
}
```

```
void Quads() // 定义四边形
{   glBegin(GL_QUADS); // 开始定义四边形
    glTexCoord2f(0, 0), glVertex2f(0, 0); // 指定纹理坐标和顶点坐标
    glTexCoord2f(1, 0), glVertex2f(1, 0);
    glTexCoord2f(1, 1), glVertex2f(1, 1);
    glTexCoord2f(0, 1), glVertex2f(0, 1);
    glEnd(); // 结束四边形定义
}
```

```
void Triangles() // 定义三角形
{   glBegin(GL_TRIANGLES); // 开始定义三角形
    glTexCoord2f(0, 0), glVertex2f(0, 0); // 指定纹理坐标和顶点坐标
    glTexCoord2f(1, 0), glVertex2f(1, 0);
    glTexCoord2f(1, 1), glVertex2f(0.5, 1); // 会产生变形
    glEnd(); // 结束三角形定义
}
```



```
void Paint() // 场景绘制函数
{   glClear(GL_COLOR_BUFFER_BIT); // 清除颜色缓存
    glLoadIdentity(); // 当前矩阵改为单位矩阵，不再有其他造型变换
    glColor3f(0.5, 0.5, 0.5); // 灰色物体
    Quads(); // 定义四边形
    glTranslatef(1, 0, 0); // 右移1
    Triangles();
    glFlush();
}
```

```
void TexImage(char *FileName) // 创建纹理图像, 使用24 bit位图文件
{   FILE *file = fopen(FileName, "rb"); // 打开文件
    fseek(file, sizeof(BITMAPFILEHEADER), SEEK_SET); // 跳过文件头
    BITMAPINFOHEADER info; // 位图信息头
    fread(&info, sizeof(BITMAPINFOHEADER), 1, file); // 获取位图信息头
    long Width = info.biWidth, Height = info.biHeight; // 图像的宽度和高度
    GLubyte Pixels[Width * Height * 3]; // 图像数据
    fread(Pixels, 1, sizeof Pixels, file); // 获取图像数据
    fclose(file); // 关闭文件
    // 二维纹理(目标纹理, 分辨率级别, 颜色分量数目,
    // 宽度, 高度, 边界宽度, 像素数据格式, 像素数据类型, 图像数据)
    glTexImage2D(GL_TEXTURE_2D, 0, 3, Width, Height, 0,
                 GL_BGR_EXT, GL_UNSIGNED_BYTE, Pixels);
}
```

```
void TexParameter() // 设置纹理参数
{   // 放大滤波，用z坐标靠近像素中心的4个纹素进行线性插值
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR);
    // 缩小滤波，用z坐标靠近像素中心的4个纹素进行线性插值
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);
}
```

```
void Hint() // 初始化，指定并启用纹理
{   TexImage("Flower.bmp"); // 创建纹理图像，使用24 bit位图文件
    TexParameter(); // 设置纹理参数
    glEnable(GL_TEXTURE_2D); // 启用二维纹理
}
```

```
int main(int argc, char *argv[])
{   glutInit(&argc, argv);
    glutInitWindowSize(512, 256); // 窗口大小(2 : 1)
    glutCreateWindow("二维纹理映射");
    Hint(); // 初始化, 指定并启用纹理
    glutDisplayFunc(Paint);
    glutReshapeFunc(Reshape);
    glutMainLoop();
}
```

2. 自动生成纹理坐标的例子

下列程序使用纹理坐标生成函数绘制一个有轮廓线的茶壶。运行结果如图8-6所示。

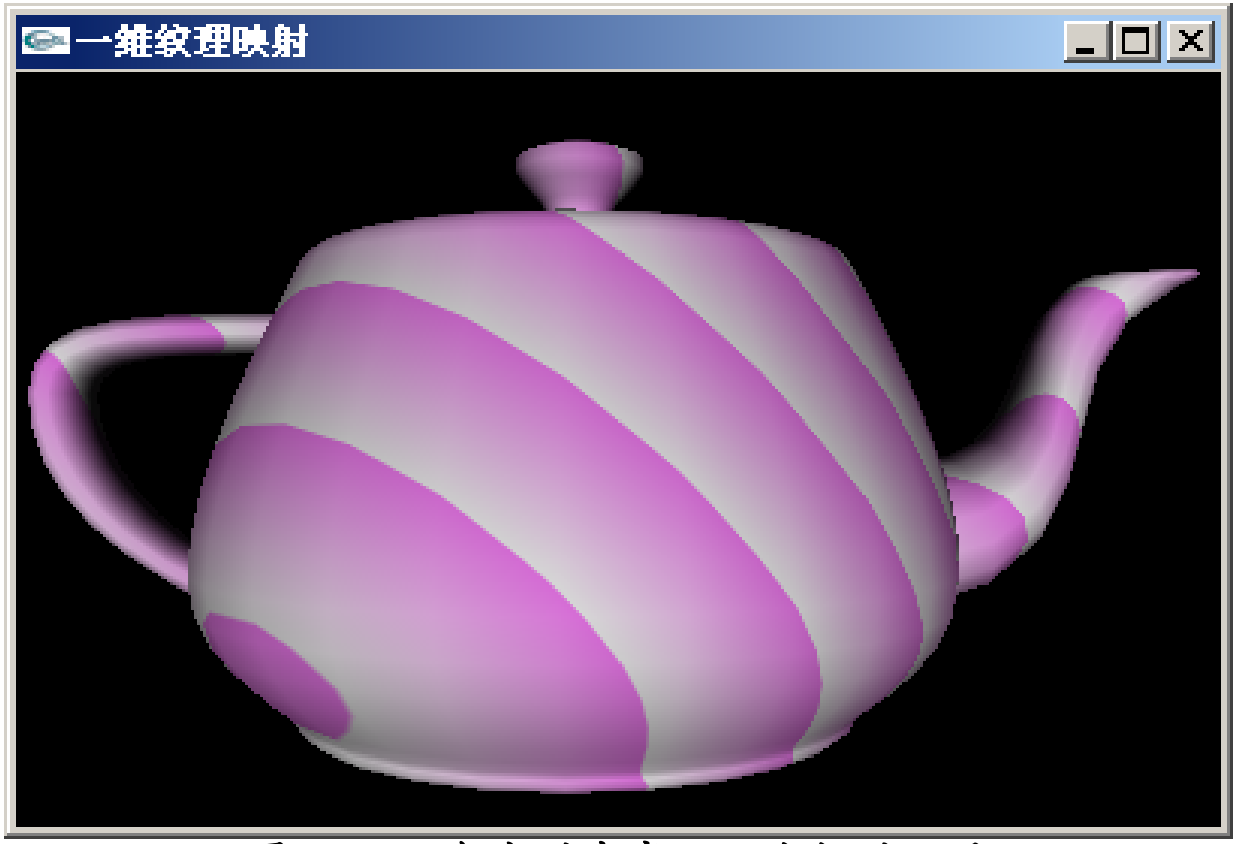


图8-6　一个自动生成纹理坐标的例子

```
// Texture1d.c
#include<gl/freeglut.h>
void Reshape(int w, int h) // 窗口变化响应函数
{   glViewport(0, 0, w, h); // 根据指定位置和大小创建视口
    glMatrixMode(GL_PROJECTION); // 投影矩阵栈
    glLoadIdentity(); // 当前矩阵改为单位矩阵，不再有其他投影变换
    gluPerspective(40, (float)w / h, 1, 1000);
    // 透视变换(视角40度，宽高比w/h，近平面1，远平面1000)
    glTranslatef(-0.075, 0, -2); // 左移0.075，远移2
    glMatrixMode(GL_MODELVIEW); // 视图造型矩阵栈
    glLoadIdentity(); // 当前矩阵改为单位矩阵，不再有其他造型变换
}
void Paint() // 场景绘制函数
{   glClear(GL_COLOR_BUFFER_BIT); // 清除颜色缓存
    glClear(GL_DEPTH_BUFFER_BIT); // 清除深度缓存
    glutSolidTeapot(0.7); // 实体茶壶
    glFlush();
}
```

```
void TexImage() // 创建纹理图像
{   // 定义一个一维纹理数据，保持红色和蓝色分量为255(MAX)，
    // 所以是渐变的紫色纹理，饱和度不断变化.
    int Width = 64; // 纹理图像宽度
    GLubyte Texture[Width][3]; // 纹理图像数据
    for(int i = 0; i < Width; ++i)
        Texture[i][0] = 255, Texture[i][1] = 255 - 2 * i, Texture[i][2] = 255;
    glTexImage1D(GL_TEXTURE_1D, 0, 3, Width, 0, GL_RGB,
                 GL_UNSIGNED_BYTE, Texture);
    // 一维纹理(目标纹理，分辨率级别，颜色分量数目，
    // 宽度，边界宽度，像素数据格式，像素数据类型，图像数据)
}
```

```
void TexParameter() // 设置纹理参数
{   glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MAG_FILTER,
                    GL_LINEAR);
    // 放大滤波, 用z坐标靠近像素中心的4个纹素进行线性插值
    glTexParameteri(GL_TEXTURE_1D, GL_TEXTURE_MIN_FILTER,
                    GL_LINEAR);
    // 缩小滤波, 用z坐标靠近像素中心的4个纹素进行线性插值
    glTexGeni(GL_S, GL_TEXTURE_GEN_MODE, GL_OBJECT_LINEAR);
    // 使用GL_OBJECT_LINEAR对应的函数自动产生纹理的S坐标
    glTexGenfv(GL_S, GL_OBJECT_PLANE, (const float[]) {1, 1, 1, 0});
    // 使用系数为{1, 1, 1, 0}的函数自动产生纹理的S坐标
}
```

```
void Hint() // 初始化(深度测试, 光照特性, 纹理特性等)
{   glEnable(GL_DEPTH_TEST); // 启用深度测试
    glEnable(GL_LIGHTING); // 激活光照
    glEnable(GL_LIGHT0); // 打开0号光源
    TexImage(); // 创建纹理图像
    TexParameter(); // 设置纹理参数
    glEnable(GL_TEXTURE_1D); // 启用一维纹理
    glEnable(GL_TEXTURE_GEN_S); // 启用纹理S坐标自动产生
}
```

```
int main(int argc, char *argv[])
{   glutInit(&argc, argv);
    glutInitWindowSize(400, 250);
    glutCreateWindow("一维纹理映射");
    Hint(); // 初始化(深度测试, 光照特性, 纹理特性等)
    glutDisplayFunc(Paint);
    glutReshapeFunc(Reshape);
    glutMainLoop();
}
```

8.6# 雾化

在OpenGL中使用雾化需要完成下列工作。

- 启用雾。
- 指定雾的颜色。
- 选取计算雾化因子的公式。
- 指定雾浓度或雾化范围（起始距离和结束距离）。

8.6.1 相关函数

1. 雾化的启用和关闭

- glEnable(GL_FOG)。启用雾化。
- glDisable(GL_FOG)。关闭雾化。

2. 指定雾的颜色

【调用形式】

- void glFogfv(GL_FOG_COLOR, GLfloat *params);
- void glFogiv(GL_FOG_COLOR, GLint *params);

【参数】params使用RGBA值指定雾的颜色，默认值为(0, 0, 0, 0)。

3. 选取计算雾化因子的公式

【调用形式】glFogi(GL_FOG_MODE, GLint param);

【参数】param可选GL_EXP(指数，默认值)、GL_EXP2(二次指数)或GL_LINEAR(线性)。

4. 指定雾浓度或雾化范围

（1）指定雾浓度。

【调用形式】

- void glFogf(GL_FOG_DENSITY, GLfloat param);
- void glFogi(GL_FOG_DENSITY, GLint param);

【参数】param是雾的浓度，用于GL_EXP和GL_EXP2，默认值为1。

（2）指定起始距离。

【调用形式】

- void glFogf(GL_FOG_START, GLfloat param);
- void glFogi(GL_FOG_START, GLint param);

【参数】param是雾化起始距离，用于GL_LINEAR，默认值为0。

（3）指定终止距离。

【调用形式】

- void glFogf(GL_FOG_END, GLfloat param);
- void glFogi(GL_FOG_END, GLint param);

【参数】param是雾化终止距离，用于GL_LINEAR，默认值为1。

8.6.2 举例

下列程序在一个窗口中使用实心方式连续绘制三个内部圆弧半径为0.2，外接球面半径为0.6的蓝色圆环，并且给出了不使用雾化、使用线性方程计算雾化和使用指数方程计算雾化的运行结果。运行结果如图8-7所示。比较采用雾化以后的结果，可以明显看出整个物体在场景中显得更加自然。

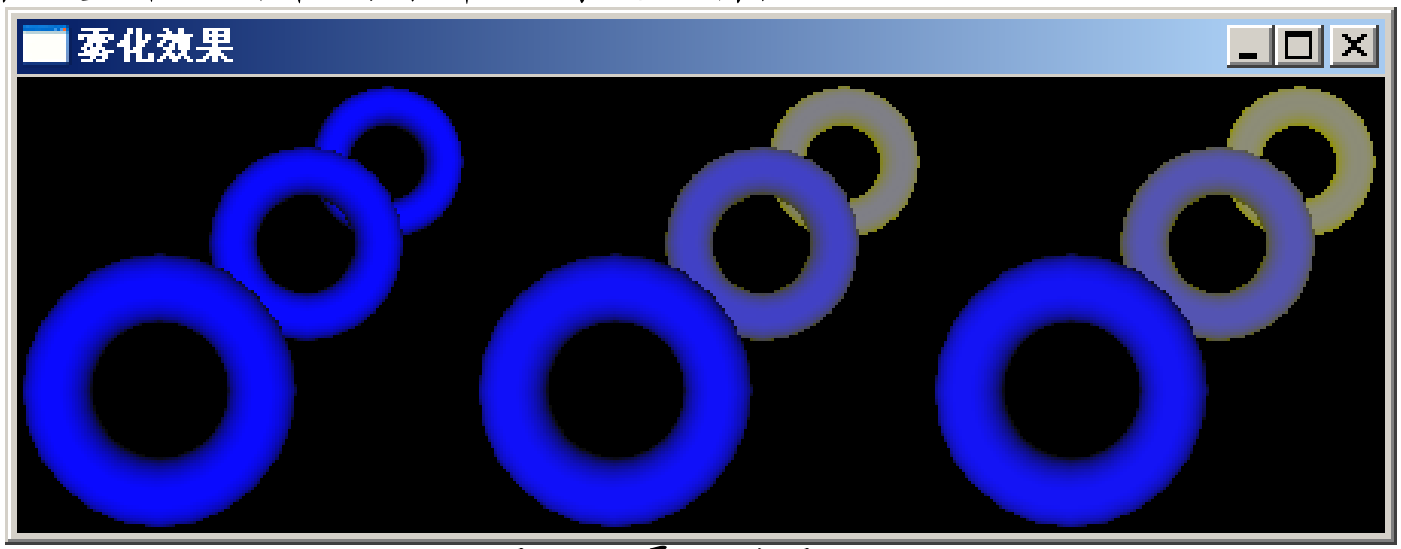


图8-7 雾化效果示例

```
// Fog.c
#include<gl/freeglut.h>
```

```
void Viewport(int x, int y, int w, int h) // 创建视口，指定投影变换
{   glViewport(x, y, w, h); // 根据指定位置和大小创建视口
    glMatrixMode(GL_PROJECTION); // 投影矩阵栈
    glLoadIdentity(); // 当前矩阵改为单位矩阵，不再有其他投影变换
    gluPerspective(30, (float)w / h, 1, 1000);
    // 透视变换(视场角30度，宽高比w/h，近平面1，远平面1000)
    glTranslatef(-0.5, -0.5, -5); // 左移0.5，下移0.5，远移5
    glMatrixMode(GL_MODELVIEW); // 视图造型矩阵栈
    glLoadIdentity(); // 当前矩阵改为单位矩阵，不再有其他造型变换
}
```

```
void Hint() // 初始化(深度测试和光照特性等)
{   glEnable(GL_LIGHTING); // 激活光照
    glEnable(GL_LIGHT0); // 打开0号光源
    glEnable(GL_DEPTH_TEST); // 启用深度测试
}
```

```
void Torus() // 定义对象(3个圆环)
{   glMaterialfv(GL_FRONT, GL_DIFFUSE, (const float[]) {0, 0, 1, 1}); // 蓝色物体
    for(int i = 0; i < 3; ++i) // 位置不同的三个圆环
    {   glutSolidTorus(0.2, 0.6, 24, 24);
        // 实体圆环(内部圆半径0.2, 外接球半径0.6, 内部圆分割数24, 圆个数24)
        glTranslatef(1, 1, -2); // 下一圆环右移1, 上移1, 远移2
    }
}
```

```
void Paint() // 场景绘制函数
{   int w = glutGet(GLUT_WINDOW_WIDTH) / 3; // 视口宽度
    int h = glutGet(GLUT_WINDOW_HEIGHT); // 视口高度
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // 清除颜色缓存和深度缓存
```

```
    Viewport(0, 0, w, h); // 左侧
    glDisable(GL_FOG); // 不使用雾化
    Torus(); // 定义对象(3个圆环)
```

```
        Viewport(w, 0, w, h); // 中间
        glFogfv(GL_FOG_COLOR, (const float[]) {1, 1, 0, 1}); // 雾的颜色(黄色)
        glFogi(GL_FOG_MODE, GL_LINEAR); // 线性变化的雾化效果
        glFogf(GL_FOG_START, 0); // 雾化起始值
        glFogf(GL_FOG_END, 8); // 雾化终止值
        glEnable(GL_FOG); // 使用雾化
        Torus();
        Viewport(2 * w, 0, w, h); // 右侧
        glFogi(GL_FOG_MODE, GL_EXP); // 指数变化的雾化效果(默认值)
        glFogf(GL_FOG_DENSITY, 0.2); // 雾浓度
        glEnable(GL_FOG); // 使用雾化
        Torus();
        glFlush();
    }
```

(1/0.2)/(1-exp(-1))≈8

```
int main(int argc, char *argv[])
{   glutInit(&argc, argv);
    glutInitWindowSize(450, 150); // 窗口大小(3 : 1)
    glutCreateWindow("雾化效果");
    Hint(); // 初始化(深度测试和光照特性等)
    glutDisplayFunc(Paint);
    glutMainLoop();
}
```

8.7 练习题

8.7.1# 基础知识题

1. 请写出OpenGL中启用点、线、面反走样的函数调用。

2. 请使用OpenGL和GLUT编写一个简单的图形程序，用于演示点的反走样效果。要求使用线段(-0.6, -0.6) ~ (0.6, 0.6)上均匀分布的5个点（含端点），点的大小为10.5像素，程序窗口的大小为(200, 200)，标题为“点的反走样”。

3. 请使用OpenGL和GLUT编写一个简单的图形程序，用于演示线段的反走样效果。其中线段的端点为(-0.6, -0.3)和(0.6, 0.3)，线宽为4.5像素，程序窗口的大小为(200, 200)，标题为“线段的反走样”。

8.7.2 程序设计题

1. 使用OpenGL和GLUT编写一个程序，用于模拟一个非常光滑的纯白球面在烈日暴晒下的效果。

2. 已知在一个空旷的场景中有一个粗糙的紫色球体，球体的右上角方向放置了一个白色的点光源，请使用OpenGL和GLUT编写一个程序模拟出球面上的光照效果（不考虑环境光）。

3#. 使用OpenGL和GLUT编写一个程序，绘制一个球体，在该球体上贴上一个渐变蓝色条纹的纹理。

8.7.3 阶段实习题

1. 完成一个简单的日地月系统演示程序。要求必须考虑太阳的自转、地球和月亮的公转和自转；能够演示地球上的白天黑夜和四季变化，以及月亮的圆缺效果。为了增强演示效果，请绘制出太阳、地球和月亮的经纬线、赤道和轴线。为了增强真实感，请在太阳、地球、月亮的表面使用合适的纹理。太阳的自转周期、地球和月亮的自转和公转周期以及地球的黄赤角等数据请到互联网上查阅。

2. 为简单日地月系统演示程序增加演示日食和月食效果的功能。要求当日食发生的时候，在地球上被月亮遮挡掉太阳光的区域（也就是地球上能够观察到日食的区域）显示一块阴影，当月食发生的时候，月亮上被地球遮挡掉太阳光的区域能够显示一块阴影。