厦門大學

计算机图形学实验

实验 2、OpenGL 绘制基础

姓　　名：黄子安

学　　号：22920212204396

学　　院：信息学院

专　　业：软件工程

年　　级：2021 级

2023 年 4 月 6 日

目录

Task1: 绘制 Sierpinski 镂垫程序

1.理解并实现课本程序

1.1 程序一

操作流程:

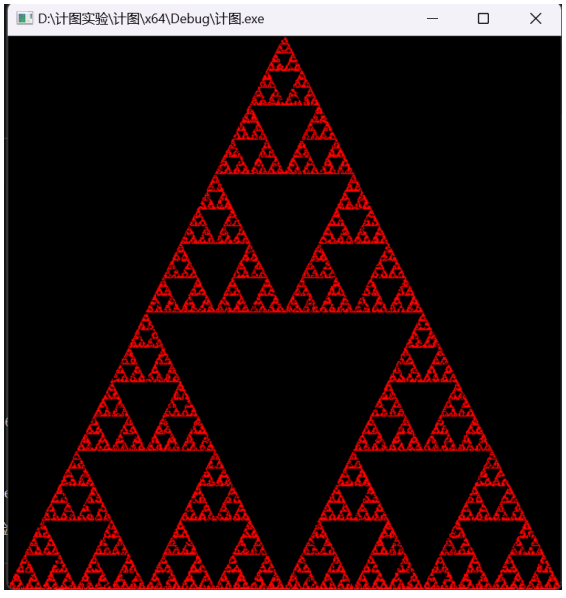
书中利用了两种方式绘制了镂垫

第一个实现 Sierpinski 镂垫程序通过循环和随机点选取的方式生成镂垫，通过随机选取顶点之后取中点的方式构造镂垫，根据分形几何的知识当循环次数够多时会生成性质稳定的三角形

代码截图：

```
void display(void)
{
    GLfloat vertices[3][2] = { {-1.0,-1.0},{1.0,-1.0},{0.0,1.0} };
    int j, k;
    GLfloat p[2] = { 0.0,0.0 };
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_POINTS);
    for (k = 0; k < 50000; k++)
    {
        j = rand() % 3;
        p[0] = (p[0] + vertices[j][0]) / 2.0;
        p[1] = (p[1] + vertices[j][1]) / 2.0;
        glVertex2fv(p);
    }
    glEnd();
    glFlush();
}
```

运行结果：



1.2 程序二

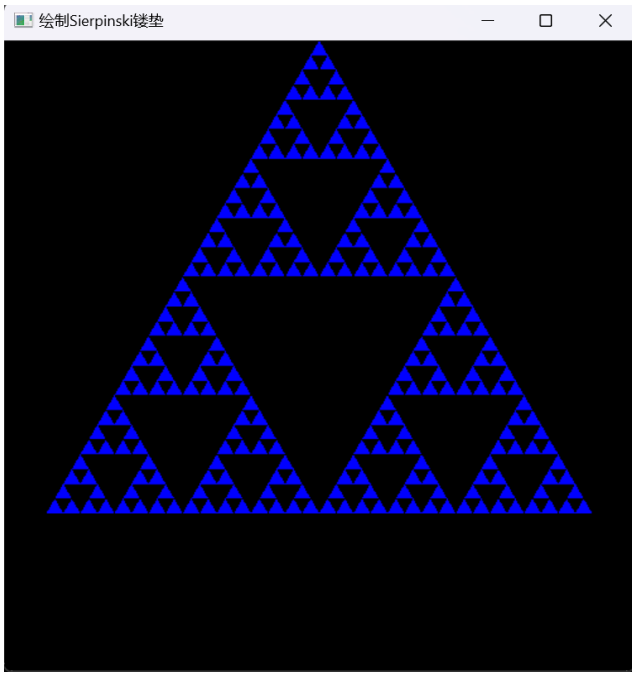
操作流程:

第二个实现 Sierpinski 镂垫程序利用了镂垫最后的形态是不断取三角形中位线的特点，直接利用构造中位线的形式生成三角形

代码截图:

```
void triangle(GLfloat *a, GLfloat *b, GLfloat *c)
{
    glVertex2fv(a);
    glVertex2fv(b);
    glVertex2fv(c);
}
void divide_triangle(GLfloat *a, GLfloat *b, GLfloat *c,int k)
{
    GLfloat ab[2], ac[2], bc[2];
    if (k)
    {
        for (int j = 0;j < 2;++j) ab[j] = (a[j] + b[j]) / 2;
        for (int j = 0;j < 2;++j) ac[j] = (a[j] + c[j]) / 2;
        for (int j = 0;j < 2;++j) bc[j] = (b[j] + c[j]) / 2;
        divide_triangle(a, ab, ac, k - 1);
        divide_triangle(c, ac, bc, k - 1);
        divide_triangle(b, bc, ab, k - 1);
    }
    else triangle(a, b, c);
}
void display()
{
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_TRIANGLES);
        divide_triangle(vertices[0],vertices[1],vertices[2],n);
    glEnd();
    glutSwapBuffers();
}
```

运行结果:



2.为不同三角形设置不同的颜色

2.1 程序一

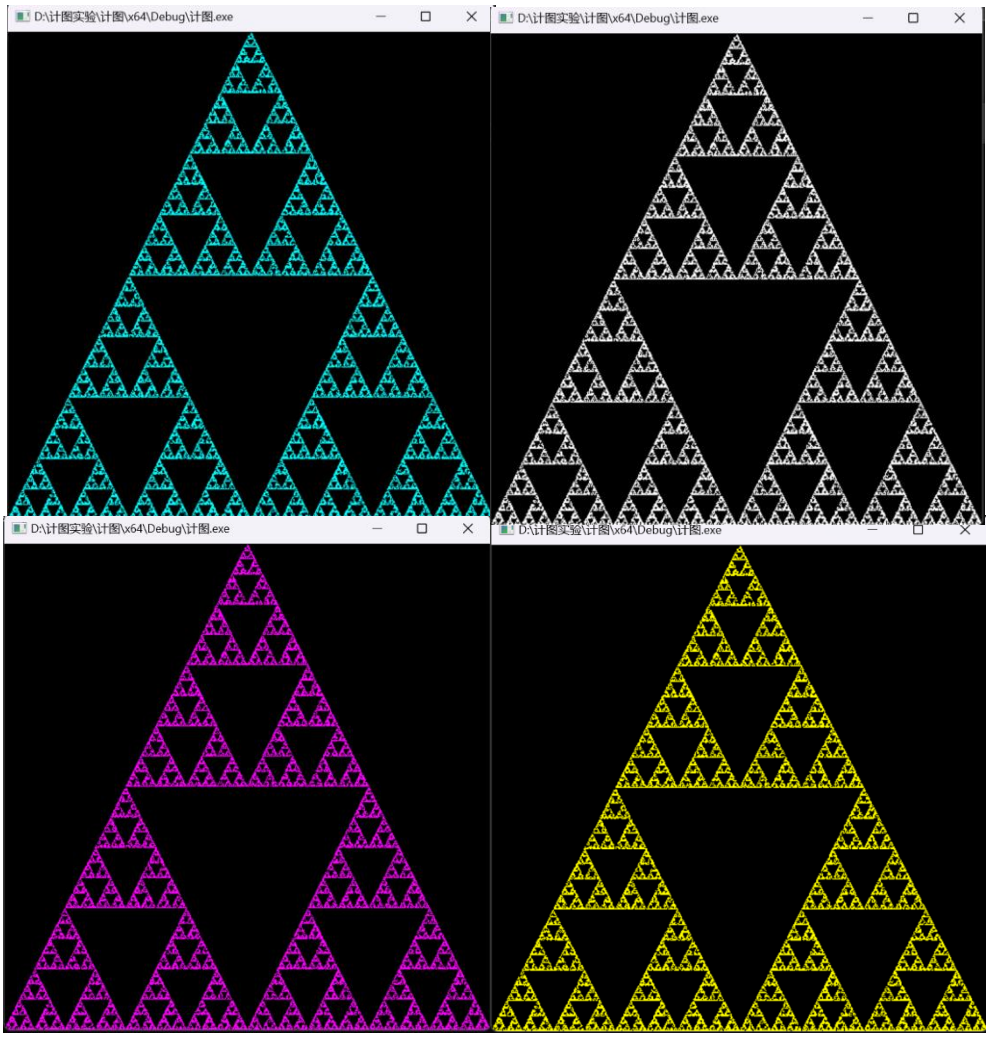
操作流程:

对于第一个利用取点生成镂垫，无法实现整个镂垫中各个三角形颜色都不相同，但是可以通过修改初始时的颜色来实现整体颜色改变，此时每个点的颜色都会发生改变，最后整个镂垫颜色也会对应发生改变

代码截图:

```
void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glColor3f(1.0, 1.0, 0.0);
}
```

运行结果:



2.2 程序二

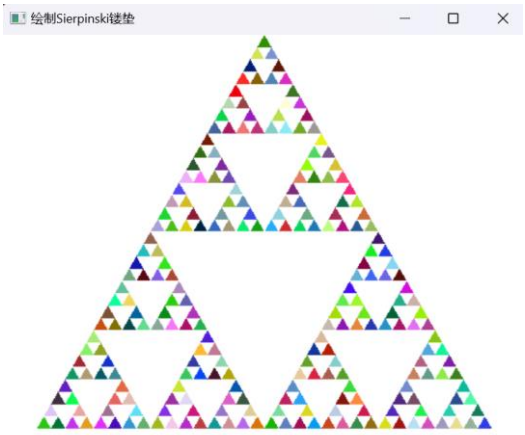
操作流程:

第二个程序因为采用递归的方式绘制三角形，因此可以在递归的终点绘制三角形时随机设置不同的颜色，使得可以最终镂垫中每个三角形的颜色都有所不同，此时运行结果中每个绘制的三角形颜色将会不同

代码截图:

```
void triangle(GLfloat *a, GLfloat *b, GLfloat *c)
{
    glColor3f(1.0f * rand() / RAND_MAX, 1.0f * rand() / RAND_MAX, 1.0f * rand() / RAND_MAX);
    glVertex2fv(a);
    glVertex2fv(b);
    glVertex2fv(c);
}
```

运行结果:



3.为镂垫生成动画

操作流程:

该任务需要实现镂垫颜色随时间发生变化，同时镂垫还需要旋转和缩放，在总体上需要注册一个 idle_func 显示回调函数，在其中设置镂垫的旋转角度和大小比例，之后在 display 函数中利用 openGL 的视物变换矩阵 glScalef，glRotatef 实现镂垫的旋转和缩放，最后在 idle_func 对应的回调函数中利用 glutPostRedisplay()来实现重新绘制,考虑到颜色闪烁太快可以使用一个时间变量适当增大时间变换的间隔

关键代码截图:

1.利用 glLoadIdentity 清除之前的变换矩阵，避免前后的矩阵叠加出现错误

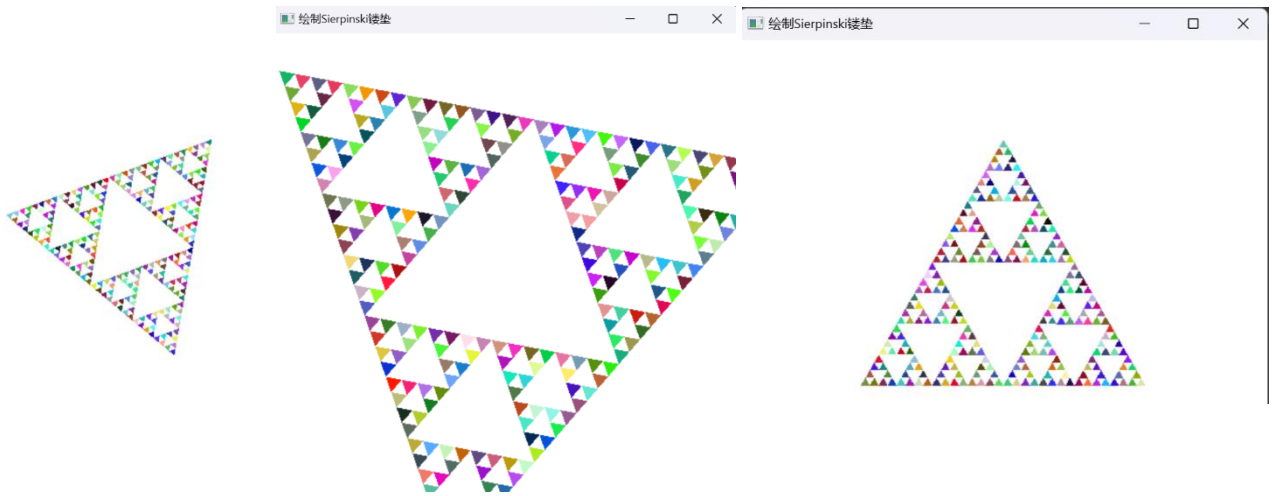
2.缩放比例 disize 利用三角函数知识可以实现镂垫较为平滑的缩放

3.可以利用双缓冲区实现动画更加平滑

```
if ((int)t%20==0)
{
    glColor3f(1.0f * rand() / RAND_MAX, 1.0f * rand() / RAND_MAX, 1.0f * rand() / RAND_MAX);
}
```

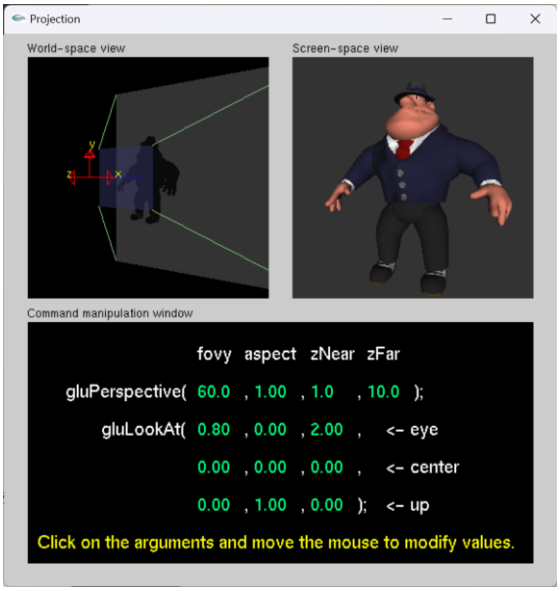
```
void display()
{
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glScalef(dsize, dsize, dsize);
    glRotatef(spin, 0, 0, 1);
    glClear(GL_COLOR_BUFFER_BIT);
    glBegin(GL_TRIANGLES);
        divide_triangle(vertices[0],vertices[1],vertices[2],n);
    glEnd();
    glutSwapBuffers();
}

void idle_func()
{
    spin += 0.015f;
    if (spin >= 360) spin -= 360;
    t += 0.005f;
    dsize = 1 + 0.5*sin(0.05*t);
    glutPostRedisplay();
}
```

运行效果:



Task2: 旋转的正方体

1.运行示例程序



思考题：

a. OpenGL 中，三维空间的坐标系是怎么样的？

在 OpenGL 中，三维空间的坐标系通常采用右手坐标系，其中 x 轴指向右侧，y 轴指向上方，z 轴是屏幕法线方向且指向屏幕外侧。

b. OpenGL 中，相机的方位是怎么样的？如何调整相机朝向呢？

相机的方位由相机位置，相机朝向和相机的向上向量三部分组成，在代码中体现为 gluLookAt 的九个参数，相机位置确定相机在世界坐标系中的位置，相机朝向决定了相机朝向哪个地方，最后相机向上向量影响摄像机镜头的自身旋转角度。

调整相机的朝向可以通过调整 gluLookAt()函数的这些参数完成，从而修改对应程序中的视物矩阵来实现调整相机的朝向。

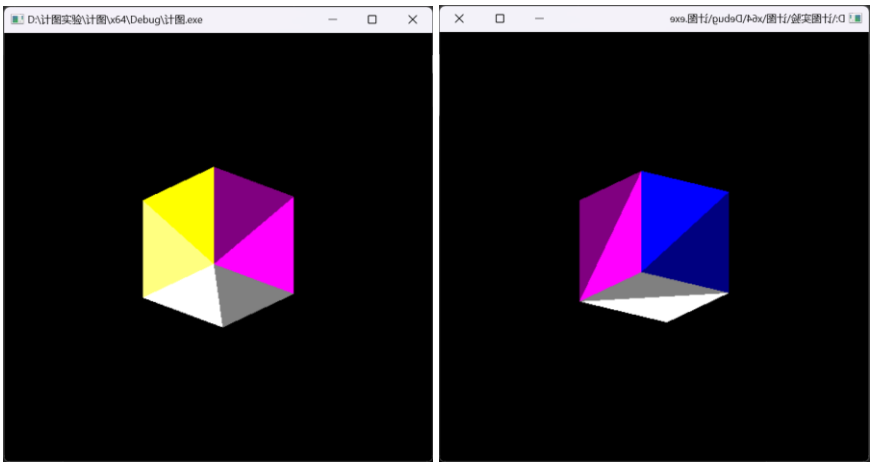
c. OpenGL 中，相机可见范围是如何设置的？

在 OpenGL 中，相机可见范围是由投影矩阵来定义的。投影矩阵可以设置透视投影（perspective projection）或正交投影（orthographic projection）：透视投影是指相机的视景体是一个锥形，体现在代码上是使用 glFrustum()函数或 gluPerspective()来更改视景体的属性，例如视景体的宽高比、近裁剪平面和远裁剪平面的位置等；而正交投影则。是指相机看到的物体大小和相机距离无关可以，用 glOrtho()函数来设置正交投影区域的属性

2. 运行提供的示例程序(exp2-2-1.cpp)

(1)深度缓冲区

比较开启/不开启深度缓冲区 glEnable(GL_DEPTH_TEST) 的效果;理解深度缓冲区的作用、用法



效果：左图为开启深度缓冲区的效果，右侧没有开启，

作用：可以看到开启深度缓冲区后实现了消隐的效果，即将不应该看到的背面给遮挡住了。即深度缓冲区会记录每个像素的深度值（z 坐标），并根据这些深度值来确

定哪些像素可见，深度较浅的像素会覆盖掉深度较深的像素点，深度检测可以达到消隐的效果，也可以提高渲染效率

用法：

1. 开启深度缓冲区：通过 glEnable(GL_DEPTH_TEST)来启用深度测试。这将开启深度缓冲区并启用深度测试功能。
2. 清除深度缓冲区：在每次渲染前需要通过 glClear(GL_DEPTH_BUFFER_BIT)来清除深度缓冲区。
3. 设置深度测试函数：通过 glDepthFunc()函数来设置深度测试函数，来确定哪些像素可见。
4. 根据点的深度来实现由系统自动实现覆盖效果

(2)正方体自行旋转

操作流程:

先修改代码实现绘制一个正方体：

创建一个点坐标结构体，含有 x、y、z 三个分量，之后创建一个顶点数组记录立方体的八个顶点，最后创建一个面数组，记录每一个由哪几个顶点构成，这样便可以在绘制时调用循环快速完成绘制

之后设置在每次 display 时利用 glRotatef 将正方体旋转一个角度，同时注册一个 glutIdleFunc 的回调函数，在其中增大旋转的角度来实现旋转效果

代码截图:

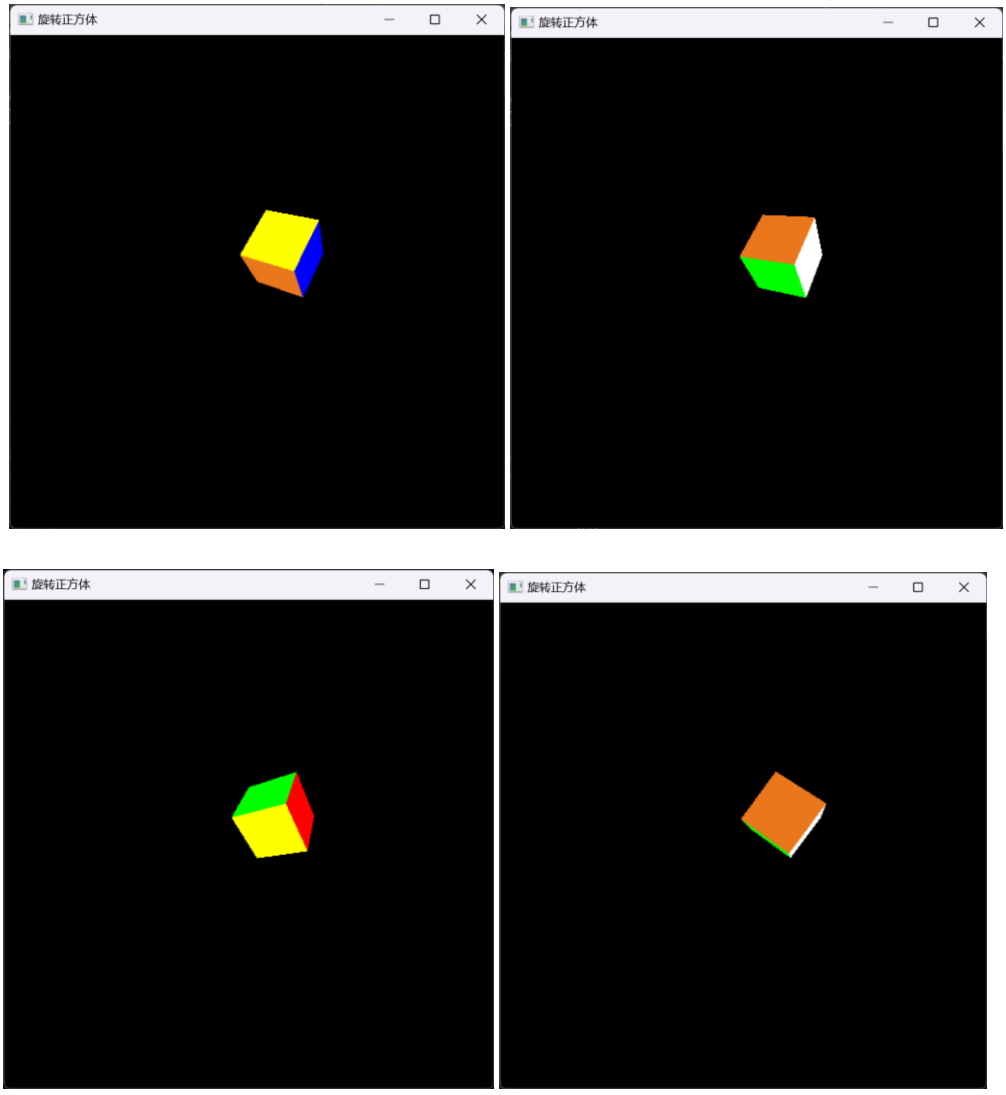
```
GLfloat color[][3] = { {1,1,0},{1,0,0},{0,1,0},{234.0/255,119.0/255,27.0/255},{0,0,1},{1,1,1} };
//顶点数组
point p[]= { {a,a,a},{a,a,-a},{-a,a,-a},{-a,a,a},
             {a,-a,a},{a,-a,-a}, {-a,-a,-a},{-a,-a,a} };
//面的顶点数组
int face[][4] = {{0,1,2,3},{0,1,5,4},{1,2,6,5},{2,3,7,6},{0,3,7,4},{4,5,6,7}};
```

```
glRotatef(80, 1, 1, 1);
glRotatef(spin, -1, 1, -1);
for (int i = 0;i < 6;++i)
{
    glColor3f(color[i][0], color[i][1], color[i][2]);
    glBegin(GL_POLYGON);
    for (int j = 0;j < 4;++j)
    {
        glVertex3f(p[face[i][j]].x, p[face[i][j]].y, p[face[i][j]].z);
    }
    glEnd();
}
```

```
void idle_func()
{
    spin += 0.02f;
    if (spin >= 360) spin -= 360;
    glutPostRedisplay();
}
```

```
glutIdleFunc(idle_func);
```

运行结果:



(3)交互相机控制

在正方体自行旋转的前提下，实现交互式的相机控制（wasd 控制相机的前进后退左右移动，qe 实现相机的升降，使用鼠标调整相机的朝向，L 锁定相机的移动和旋转）

操作流程:

I.相机移动:

先切换矩阵模式为投影矩阵，之后设置对应的 gluLookAt 参数，适当调整 gluPerspective 的参数使图像位于窗口的合适位置

之后注册 glutKeyboardFunc 的回调函数，根据输入的值改变对应的相机位置坐标来实现相机的移动

代码截图:

```
void key_func(unsigned char key, int x, int y)
{
    GLfloat movespeed = 0.25f;
    if (enable_camera_move)
    {
        switch (key)
        {
        case 27:exit(0);break;
        case 's':cameraZ += movespeed;break;
        case 'w':cameraZ -= movespeed;break;
        case 'a':cameraX -= movespeed;break;
        case 'd':cameraX += movespeed;break;
        case 'q':cameraY += movespeed;break;
        case 'e':cameraY -= movespeed;break;
        default:break;
        }
    }
    if (key == 'l') enable_camera_move=!enable_camera_move;
    glutPostRedisplay();
}
```

```
GLfloat cameraX = 0.0f, cameraY = 0.0f, cameraZ = 5.0f;
GLfloat cameraAngleFai = pi / 2, cameraAngleTheta = pi;
GLfloat upx = 0.0f, upy = 1.0f, upz = 0.0f;

GLint mouseX, mouseY;
bool enable_camera_move = true;
```

```
glutKeyboardFunc(key_func);
```

II.鼠标调整:

鼠标控制视角旋转可以通过修改 gluLookAt 的中间三个参数完成，考虑到视角转动和相机位置的关系，这里三个参数设置为相机的位置向量与一个球坐标系下视角转动向量的和，以此实现更好的旋转效果

通过 openGL 的 glutPassiveMotionFunc 函数可以实时获取鼠标光标在窗口中的坐标，利用全局变量保存这些坐标之后在绘制时修改视角向量的方位角和俯仰角来实现视角转动

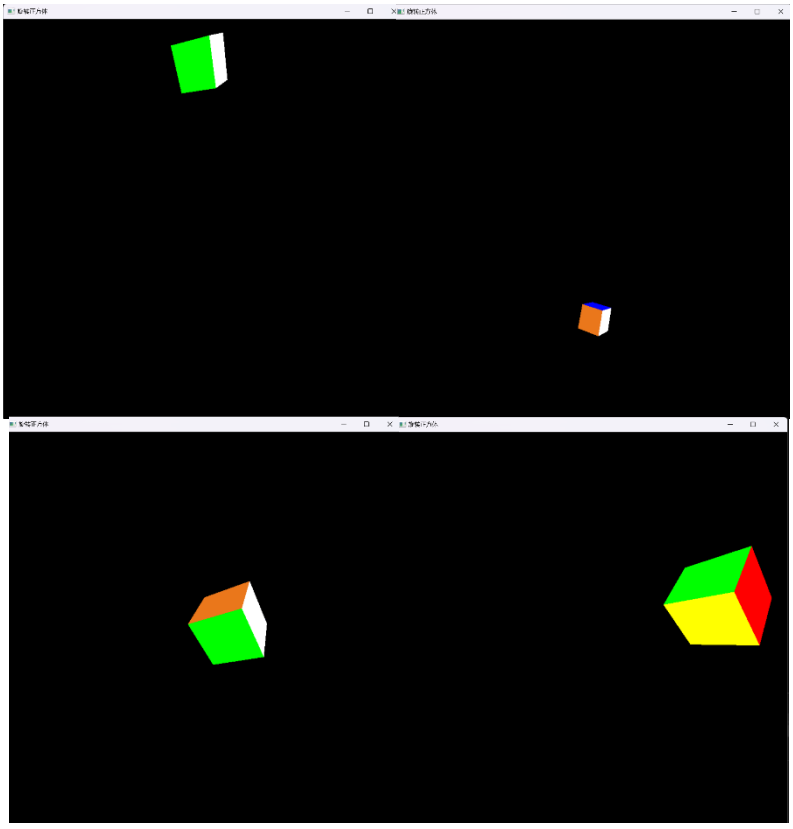
同时可以利用函数 glutWarpPointer(windows_width / 2, windows_height / 2) 实现在窗口生成时将鼠标光标的位置移到窗口中间，避免窗口生成时寻找不到物体

代码截图:

```
void motion_func(int x, int y)
{
    if (enable_camera_move)
    {
        cameraAngleTheta = pi - pi / 4 * (x - windows_width / 2) / (windows_width / 2);
        cameraAngleFai = pi / 2 + pi / 4 * (y - windows_height / 2) / (windows_height / 2);
    }
}
```

```
gluLookAt(cameraX, cameraY, cameraZ,
          cameraX + sin(cameraAngleTheta) * sin(cameraAngleFai),
          cameraY + cos(cameraAngleFai),
          cameraZ + cos(cameraAngleTheta) * sin(cameraAngleFai),
          upx, upy, upz);
```

效果截图:



思考题：

如何实现前后面的遮挡？你需要先自行了解一下深度缓冲区和深度测试的作用。

搜索：

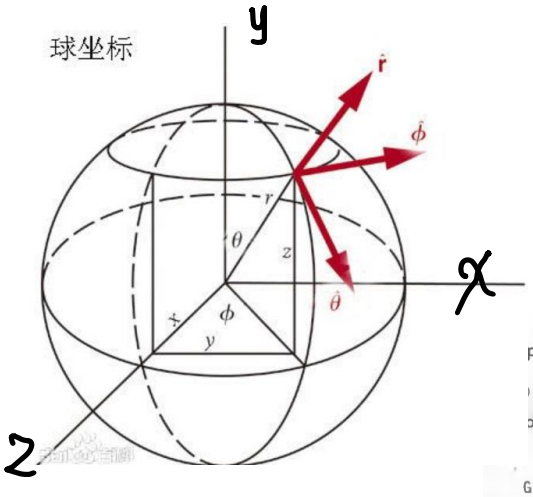
openGL 实现前后面的遮挡主要有两种方式：

一种是利用深度检测，即通过计算每一个像素点的深度值之后存入深度缓冲区，再在渲染的时候根据像素点的深度值，深度浅的像素覆盖掉深度较深的像素，以此来实现消隐的效果

另一种是利用 openGL 的 CullFace 功能，该方法会计算对应面的法向量，利用该法向量和相机到这个面的向量的数量积的正负判断该面是否会被隐藏

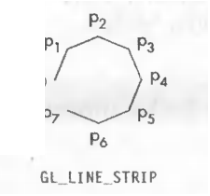
Task3: 线框球体的绘制

1.绘制一个线框球体



操作流程：

根据线框的图形样式可知线框可以由若干条经线和纬线组成，绘制经纬线时为了方便起见可以采用球坐标的形式进行绘制，绘制单条经纬线时采用图元 GL_LINE_STRIP 利用折线段近似代替曲线.



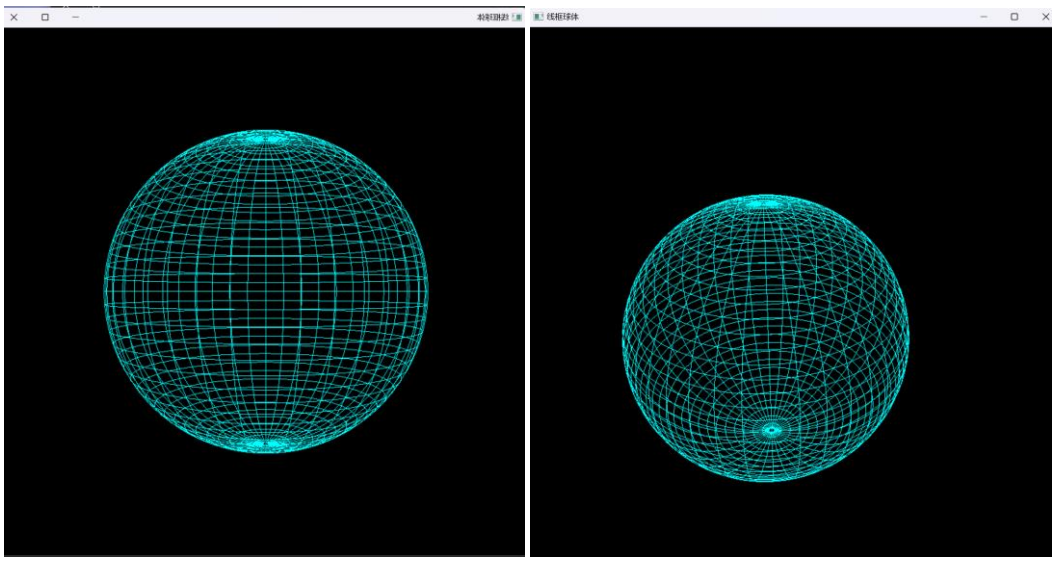
$$\begin{cases} x = rsin(\theta)sin(\varphi) \\ y = rcos(\varphi) \\ z = rcos(\theta)sin(\varphi) \end{cases}$$

关键代码截图：

```
int nw = 40;//nw表示纬线数量
int nj = 40;//hj表示经线数量
//绘制纬线
for (int i = 0;i <= nw;++i)
{
    GLfloat  fai= pi*i / nw;
    glBegin(GL_LINE_STRIP);
    int nnw = 180;//nnw表示一条纬线上的折线段数量
    for (int j = 0;j <= nnw;++j)
    {
        GLfloat theta = 2*pi*j / nnw;
        GLfloat x = r * sin(theta) * sin(fai);
        GLfloat y = r * cos(fai);
        GLfloat z = r * cos(theta) * sin(fai);
        glVertex3f(x, y, z);
    }
    glEnd();
}
//绘制经线
for (int i = 0;i <= nj;++i)
{
    GLfloat  theta = 2.0 * pi * i / nj;
    glBegin(GL_LINE_STRIP);
    int nnj = 180;//nnw表示一条经线上的折线段数量
    for (int j = 1;j < nnj;++j)
    {
        GLfloat fai = pi * j / nnj;
        GLfloat x = r * sin(theta) * sin(fai);
        GLfloat y = r * cos(fai);
        GLfloat z = r * cos(theta) * sin(fai);
        glVertex3f(x, y, z);
    }
    glEnd();
```

运行效果：

实际运行效果如图一所示，图二为展示效果进行了一定的旋转操作方便观察球框的另外部分



2、添加动画效果，让球体绕圆心旋转

操作流程：

同理于之前的实验步骤，利用 glRotatef 即可完成对应的操作，同理切换为视物矩阵并且要重新设置为单位阵，之后注册 glutIdleFunc 回调函数增大每次绘制时角度即可实现球体围绕球心旋转

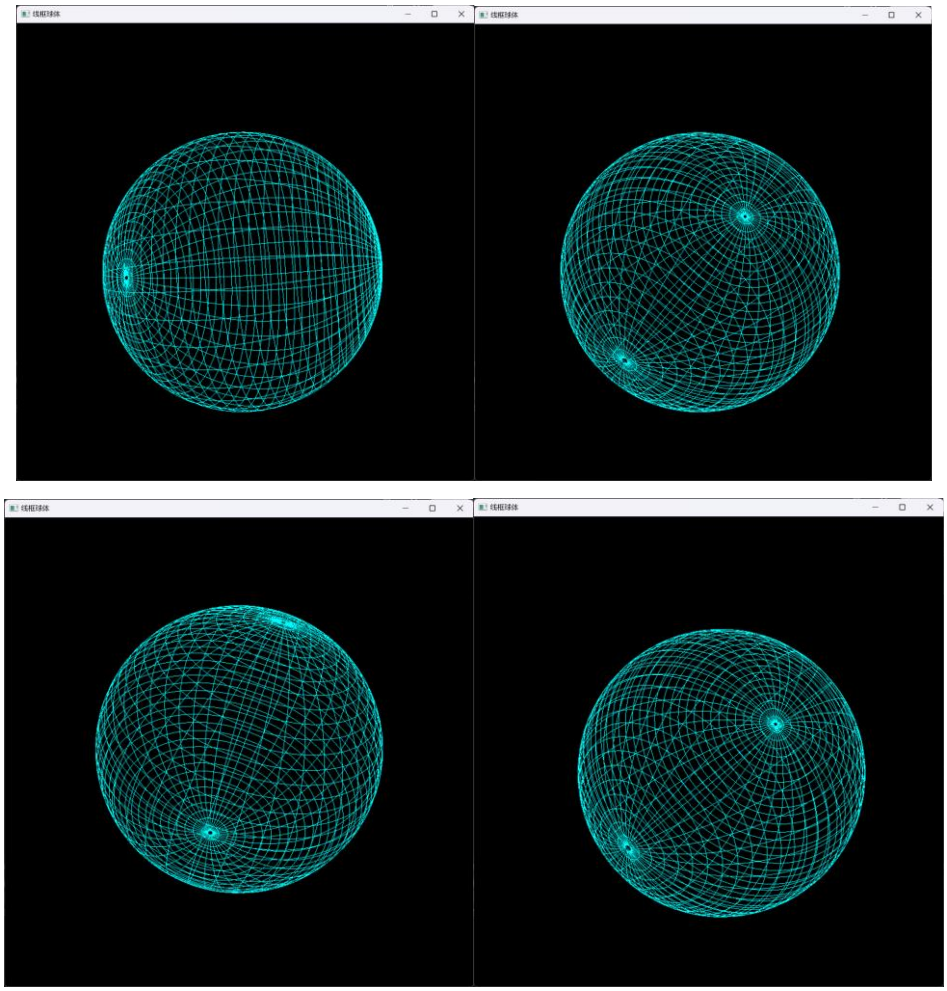
代码截图：

```
void idle_func()
{
    spin += rotatev;
    if (spin >= 360) spin -= 360;
    glutPostRedisplay();
}
```

```
glRotatef(spin, 1, 1, 0);
```

```
glutIdleFunc(idle_func);
```

运行结果:



3、保持比例不变

操作流程:

注册一个 glutReshapeFunc 的回调函数，在窗口大小改变后获取窗口的高和宽，因为在 glutIdleFunc 中会一直重新更改投影矩阵，所以不能在 reshape_func 中直接修改视锥体的属性，而是要更改 glutIdleFunc 的投影矩阵信息。使用一个全局变量记录窗口的宽和高，之后在 display 里面根据这两个属性更改视锥体的宽高比，在每一次被 glutIdleFunc 调用时都按照最新的窗口大小绘制。

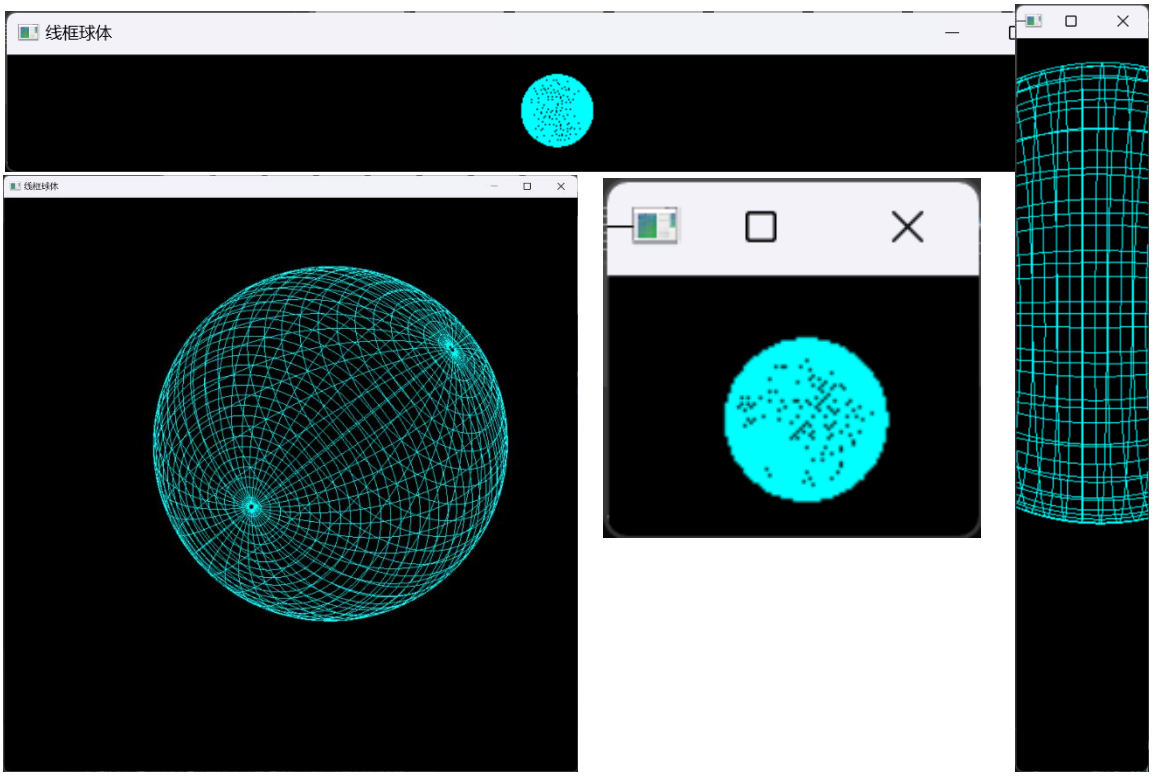
代码截图：

```
GLint windows_width = 800;
GLint windows_height = 800;
```

```
void reshape(int w, int h)
{
    windows_height = h;
    windows_width = w;
    glutPostRedisplay();
}
```

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glViewport(0, 0, windows_width, windows_height);
gluPerspective(60, windows_width * 1.0 / windows_height, 0.1, 50);
glMatrixMode(GL_MODELVIEW);
```

运行结果：



4、添加交互式的相机控制

操作流程:

完全同理于任务二，利用 glutKeyboardFunc 获取按下的键和 glutPassiveMotionFunc 获取当前鼠标的坐标，之后根据这些属性修改 gluLookAt 的参数完成相机的交互式控制

关键代码:

```
void key_func(unsigned char key, int x, int y)
{
    GLfloat movespeed = 0.25f;
    if (enable_camera_move)
    {
        switch (key)
        {
        case 27:exit(0);break;
        case 's':cameraZ += movespeed;break;
        case 'w':cameraZ -= movespeed;break;
        case 'a':cameraX -= movespeed;break;
        case 'd':cameraX += movespeed;break;
        case 'q':cameraY += movespeed;break;
        case 'e':cameraY -= movespeed;break;
        default:break;
        }
    }
    if (key == '1') enable_camera_move=!enable_camera_move;
    glutPostRedisplay();
}
```

```
GLfloat cameraX = 0.0f, cameraY = 0.0f, cameraZ = 5.0f;
GLfloat cameraAngleFai = pi / 2, cameraAngleTheta = pi;
GLfloat upx = 0.0f, upy = 1.0f, upz = 0.0f;

GLint mouseX, mouseY;
bool enable_camera_move = true;
```

```
glutKeyboardFunc(key_func);
```

```
void motion_func(int x, int y)
{
    if (enable_camera_move)
    {
        cameraAngleTheta = pi - pi / 4 * (x - windows_width / 2) / (windows_width / 2);
        cameraAngleFai = pi / 2 + pi / 4 * (y - windows_height / 2) / (windows_height / 2);
    }
}
```

```
gluLookAt(cameraX, cameraY, cameraZ,
          cameraX + sin(cameraAngleTheta) * sin(cameraAngleFai),
          cameraY + cos(cameraAngleFai),
          cameraZ + cos(cameraAngleTheta) * sin(cameraAngleFai),
          upx, upy, upz);
```

运行结果:

图三为将相机移动到球框内部