绘制太阳系

一、实验任务

绘制出一个太阳系：

要求：

1. 有详细的计算步骤
2. 至少包含太阳、地球和月亮
3. 用 OpenGL 进行绘制

Bonus：

1. 用代码实现出可执行的实例
2. 绘制出行星的轨道

二、原理和分析

1. OpenGL 材质和光照

OpenGL 在处理光照时把光照系统分为三部分，分别是光源、材质和光照模型。

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，

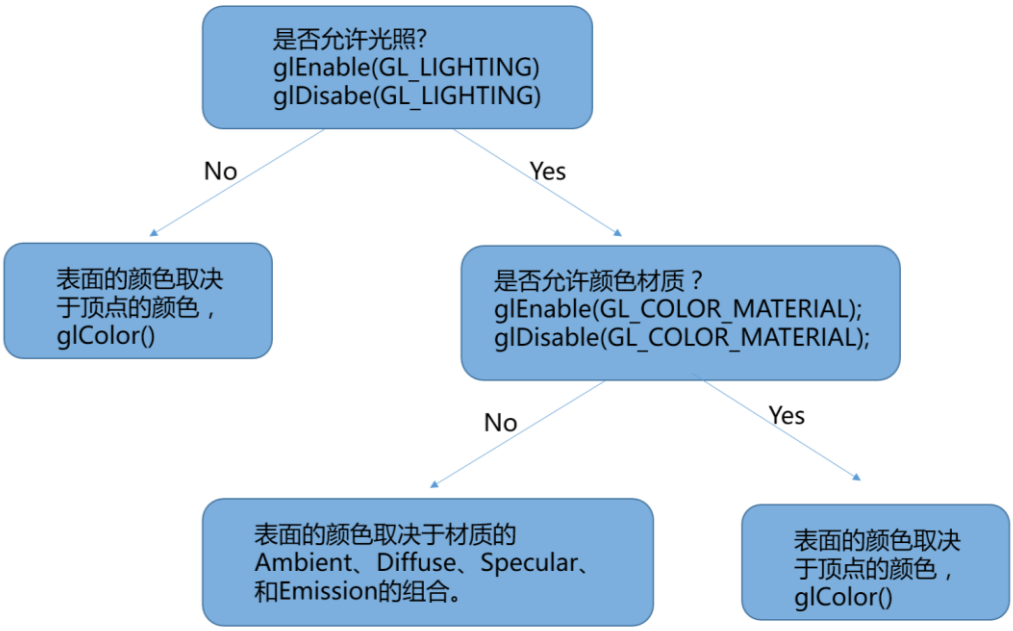
使用 glMaterial*函数可设置材质的属性，

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。

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。

OpenGL 场景中模型颜色的产生，大致为如下的流程图所描述：



光源设置

设置**环境光**：glLightfv(GL_LIGHT0,GL_AMBIENT,ambientLight);

设置**漫射光成分**：glLightfv(GL_LIGHT0,GL_DIFFUSE,DiffuseLight)

设置**镜面光成分**：glLightfv(GL_LIGHT0,GL_SPECULAR,SpecularLight);

光源的属性 GL_SPECULAR 影响镜面反射区域的颜色，一般物体的镜面反射区域的颜色为入射光线的颜色，要实现真实感，应该将它的值设置成与 GL_DIFFUSE 相同。

设置**光源的位置**：glLightfv(GL_LIGHT0, GL_POSITION, sun_light_position);

GL_POSITION 属性表示光源所在的位置。由四个值（X, Y, Z, W）表示。

方向性光源：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。通常，太阳可以近似的被认为是方向性光源。

位置性光源：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。

定位光源需要对其发射的光进行衰减，可以设置各种衰减因子。环境光，散射光和镜面反射光的贡献都是衰减的，只有发射光和全局环境光不会衰减。

材质设置

OpenGL 用材料对光的红、绿、蓝三原色的反射率来近似定义材料的颜色。像光源一样，材料颜色也分成环境、漫反射和镜面反射成分，它们决定了材料对环境光、漫反射光和镜面反射光的反射程度。

在进行光照计算时，材料对环境光的反射率与每个进入光源的环境光结合，对漫反射光的反射率与每个进入光源的漫反射光结合，对镜面光的反射率与每个进入光源的镜面反射光结合。对环境光与漫反射光的反射程度决定了材料的颜色，并且它们很相似。对镜面反射光的反射率通常是白色或灰色（即对镜面反射光中红、绿、蓝的反射率相同）。镜面反射高光最亮的地方将变成具有光源镜面光强度的颜色。

材质的颜色与光源的颜色有些不同。对于光源，R、G、B 值等于 R、G、B 对其最大强度的百分比。若光源颜色的 R、G、B 值都是 1.0，则是最强的白光；若值变为 0.5，颜色仍

为白色，但强度为原来的一半，于是表现为灰色；若 R = G = 1.0，B = 0.0，则光源为黄色。对于材质，R、G、B 值为材质对光的 R、G、B 成分的反射率。比如，一种材质的 R = 1.0、G = 0.5、B = 0.0，则材质反射全部的红色成分，一半的绿色成分，不反射蓝色成分。也就是说，若 OpenGL 的光源颜色为（LR、LG、LB），材质颜色为（MR、MG、MB），那么，在忽略所有其他反射效果的情况下，最终到达眼睛的光的颜色为（LR*MR、LG*MG、LB*MB）指定了图元的法线之后，我们还需要为其指定相应的材质以决定物体对各种颜色的光的反射程度，这将影响物体表现为何种颜色。

函数 glMaterialfv(GL_FRONT,GL_DIFFUSE,@Diffuse);可以设定物体的材质属性。

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。

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。

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。

2. OpenGL 中的坐标系

OpenGL 中总共分为 5 个空间：

① 局部空间(Local Space，或者称为物体空间(Object Space))

② 世界空间(World Space)

③ 观察空间(View Space，或者称为视觉空间(Eye Space))

④ 裁剪空间(Clip Space)

⑤ 屏幕空间(Screen Space)

世界坐标系

世界坐标系始终是固定不变的。OpenGL 中使用右手坐标。进行旋转操作时需要指定的角度θ的方向则由右手法则来决定，即右手握拳，大拇指直向某个坐标轴的正方向，那么其余四指指向的方向即为该坐标轴上的θ角的正方向（即θ角增加的方向）

对象坐标系

对象坐标系是对象在被应用任何变换之前的初始位置和方向所在的坐标系，也就是当前绘图坐标系。该坐标系不是固定的，且仅对该对象适用。在默认情况下，该坐标系与世界坐标系重合。这里能用到的函数有 glTranslatef()，glScalef()，glRotatef()，当用这些函数对当前绘图坐标系进行平移、伸缩、旋转变换之后， 世界坐标系和当前绘图坐标系不再重合。改变以后，再用 glVertex3f()等绘图函数绘图时，都是在当前绘图坐标系进行绘图，所有的函数参数也都是相对当前绘图坐标系来讲的。

观察坐标系

GL_MODELVIEW 矩阵是模型变换矩阵和视变换矩阵的组合(Mview*Mmodel)，并不存在单独的模型变换(Model)和视点变换(View)。所以使用 GL_MODELVIEW 矩阵就可以使对象从对象坐标系转换到观察坐标系。

默认情况下，眼坐标系与世界坐标系也是重合的。使用函数 gluLookAt()则可以指定眼睛(相机)的位置和眼睛看向的方向。该函数的原型如下:

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

函数参数中，点(eyex, eyey, eyez)代表眼睛所在位置；点(centerx, centery,centerz)代表眼睛看向的位置；向量(upx, upy, upz)代表视线向上方向，其中视点和参考点的连线与视线向上方向要保持垂直关系。只需控制这三个量，便可定义新的视点。

裁剪坐标系

观察坐标系到裁剪坐标是通过投影完成的。眼坐标通过乘以 GL_PROJECTION 矩阵变成了裁剪坐标。GL_PROJECTION 矩阵定义了视景体(viewing volume)，即确定哪些物体位于视野之内，位于视景体外的对象会被剪裁掉。除了视景体，投影变换还定义了顶点是如何投影到屏幕上的，是透视投影(perspective projection)还是正交投影(orthographic projection)。

屏幕坐标系

屏幕上的设备坐标称为屏幕坐标。设备坐标又称为物理坐标,是指输出设备上的坐标。设备坐标用对象距离窗口左上角的水平距离和垂直距离来指定对象的位置,是以像素为单位来表示的,设备坐标的 X 轴向右为正,Y 轴向下为正,坐标原点位于窗口的左上角。

从NDC坐标到屏幕坐标基本上是一个线性映射关系。通过对NDC坐标进行视口变换得到。这时候就要用到函数 glViewport()，该函数用来定义渲染区域的矩形，也就是最终图像映

射到的区域。

3. 键盘和鼠标事件

鼠标事件

检测鼠标 clicks 函数：

void glutMouseFunc(void(*func)(int button,int state,int x,int y));

它在程序初始化阶段被调用。函数一共有 4 个参数。第一个参数表明哪个鼠标键被按下或松开，这个变量可以是下面的三个值中的一：GLUT_LEFT_BUTTON，GLUT_MIDDLE_BUTTON，GLUT_RIGHT_BUTTON。第二个参数表明，函数被调用发生时，鼠标的状态，也就是是被按下，或松开，可能取值如下：GLUT_DOWN，GLUT_UP。

当函数被调用时，state 的值是 GLUT_DOWN，那么程序可能会假定将会有个 GLUT_UP 事件，甚至鼠标移动到窗口外面，也如此。然而，如果程序调用 glutMouseFunc 传递 NULL 作为参数，那么 GLUT 将不会改变鼠标的状态。剩下的两个参数（x, y）提供了鼠标当前的窗口坐标（以左上角为原点）。

检测动作（motion）函数

GLUT 提供鼠标 motion 检测能力。有两种 GLUT 处理的 motion：active motion 和 passive motion。Active motion 是指鼠标移动并且有一个鼠标键被按下。Passive motion 是指当鼠标移动时，并有没鼠标键按下。如果一个程序正在追踪鼠标，那么鼠标移动期间，每一帧将产生一个结果。

函数原型：

void glutMotionFunc(void(*func)(int x,int y));

void glutPassiveMotionFunc(void (*func)(int x,int y));

参数：

Func：处理各自类型 motion 的函数名。

处理 motion 的参数函数的参数（x,y）是鼠标在窗口的坐标。以左上角为原点。

键盘事件：

GLUT 提供了两个函数为这个键盘消息注册回调。

glutKeyboardFunc：告诉窗口系统，哪一个函数将会被调用来处理普通按键消息。

普通键是指字母，数字，和其他可以用 ASCII 代码表示的键。

函数原型如下：

void glutKeyboardFunc(void(*func)(unsigned char key,int x,int y));

参数：

func: 处理普通按键消息的函数的名称。如果传递 NULL 则表示 GLUT 忽略普通按键消息。

这个作为 glutKeyboardFunc 函数参数的函数需要有三个形参。第一个表示按下的键的 ASCII 码，其余两个提供了，当键按下时当前的鼠标位置。鼠标位置是相对于当前客户窗口的左上角而言的。

glutSpecialFunc：处理普通按键消息的函数

函数原型如下：

void glutSpecialFunc(void (*func)(int key,int x,int y));

参数：

func: 处理特殊键按下消息的函数的名称。传递 NULL 则表示 GLUT 忽略特殊键消息。

三、图形绘制

1. 创建行星类

```
class Star {
private:
    int ID;

    double radius;            //半径

    double rotation_period;
    double rotation_angle;
    double revolution_radius;
    double revolution_period;
    double revolution_angle;
    double angle = 0;
    double rotationAngle = 0;

public:
    GLfloat rotaVector[NUMBER][3];
public:

    //名称，半径，自转周期，自转角度，公转半径，公转周期，公转角度

    Star(int, double, double, double, double, double, double,double);
    Star();

    //设置行星的材质

    void Mertial(GLfloat*, GLfloat*, GLfloat*, GLfloat*, GLfloat);


    void DrawStar(bool run);                    //绘制行星

    void DrawStaellite(Star& Planet,bool run);      //绘制卫星

    void DrawOrbit();                           //绘制轨道

    void DrawSatelliteOrbit(Star& Planet);          //绘制卫星轨道

    void SetValue(GLfloat* , GLfloat , GLfloat , GLfloat );


    void DrawRing(GLfloat R,GLfloat Width);         //绘制行星环

};
```

创建一个类用于储存星球的半径，自转周期，自转角度，公转半径，公转周期，公转角度等。

定义关于行星运动的方法等。

2. 行星运动

由于矩阵乘法的性质和 OpenGL 中的设定，实际绘制的顺序和绘制时调用函数的顺序是相反的。由绘制卫星的例子描述 OpenGL 中的绘制过程。

```
//绘制卫星时必须在行星后面，因为要用到行星的参数
void Star::DrawStaellite(Star& Planet,bool run)
{
    if (run)
    {
        //计算卫星公转角度
        if (this->revolution_period != 0)
            this->angle += (float)(1 / this->revolution_period);
        while (this->angle >= 360)
            this->angle = this->angle - 360;

        //计算卫星自转角度
        bool flag = (rotation_period < 0) ? false : true;
        if (this->rotation_period != 0)
            this->rotationAngle += (float)(1 / this->rotation_period *
flag);
        while (this->rotationAngle >= 360)
            this->rotationAngle = this->rotationAngle - 360;
    }

    glPushMatrix();

    //行星公转轨道倾角
    glRotatef(Planet.revolution_angle,       rotaVector[Planet.ID][0],
rotaVector[Planet.ID][1], rotaVector[Planet.ID][2]);
    glRotatef(Planet.angle, 0.0f, 1.0f, 0.0f);          //行星公转

    glTranslatef(Planet.revolution_radius, 0.0f, 0.0f);//行星公转半径
```

```
    //卫星公转轨道倾角
    glRotatef(this->revolution_angle,         rotaVector[this->ID][0],
rotaVector[this->ID][1], rotaVector[this->ID][2]);
    glRotatef(angle, 0.0f, 1.0f, 0.0f);                //卫星公转

    glTranslatef(this->revolution_radius, 0.0f, 0.0f); //卫星公转半径
    glutSolidSphere(this->radius, 80.0f, 80.0f);

    glPopMatrix();
}
```

首先计算出卫星本次公转和自转的角度位置。

随后调用 glPushMatrix();函数将矩阵压栈。

由于顺序相反，故调用函数顺序从下向上观察：

首先绘制球体：

```
glutSolidSphere(this->radius, 80.0f, 80.0f);
```

随后将卫星平移至公转半径位置：

```
glTranslatef(this->revolution_radius, 0.0f, 0.0f);
```

然后根据计算出的角度将伟星管旋转至公转位置：

```
glRotatef(angle, 0.0f, 1.0f, 0.0f);
```

由于卫星具有公转倾角，故须将其公转进行旋转：

```
glRotatef(this->revolution_angle,rotaVector[this->ID][0],
rotaVector[this->ID][1], rotaVector[this->ID][2]);
```

由于卫星围绕行星公转，随后将其平移至行星公转的半径处：

```
glTranslatef(Planet.revolution_radius, 0.0f, 0.0f);
```

随后将其旋转行星公转的角度：

```
glRotatef(Planet.angle, 0.0f, 1.0f, 0.0f);
```

由于行星公转同时具有轨道倾角，故将其旋转行星公转的轨道倾角：

```
glRotatef(Planet.revolution_angle,rotaVector[Planet.ID][0],
rotaVector[Planet.ID][1], rotaVector[Planet.ID][2]);
```

3. 行星材质

```
void Star::Mertial(GLfloat* emission,GLfloat* ambient, GLfloat* diffuse, GLfloat* specular, GLfloat shininess)
{
    glMaterialfv(GL_FRONT, GL_EMISSION, emission);          //自己发光

    glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);            //环境光

    glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuse);            //漫反射

    glMaterialfv(GL_FRONT, GL_SPECULAR, specular);          //镜面反射

    glMaterialf(GL_FRONT, GL_SHININESS, shininess);         //镜面指数
}
```

行星的材质由传入的参数设定，每个行星需要在其绘制时传入相应的材质的参数。

4. 轨道绘制

```
//绘制公转轨道
void Star::DrawOrbit()
{
    glPushMatrix();
    //公转轨道倾角
    glRotatef(this->revolution_angle,          rotaVector[this->ID][0], rotaVector[this->ID][1], rotaVector[this->ID][2]);
    glDisable(GL_LIGHTING);
    glColor3f(1.0f,1.0f, 1.0f);                          //设置线段绘制颜色
    GLfloat r = this->revolution_radius;
    GLfloat x1, z1, x2, z2, y = 0;
    int NUM = (((int)r / 8000) + 1 ) * 360;
    for (int i = 0; i < NUM; i += 2)
    {
        x1 = cos(1.0 * i / NUM * 2 * M_PI) * r;
        z1 = sin(1.0 * i / NUM * 2 * M_PI) * r;
        x2 = cos(1.0 * (i+1) /NUM * 2 * M_PI) * r;
        z2 = sin(1.0 * (i+1) / NUM * 2 * M_PI) * r;
        glBegin(GL_LINES);
        glVertex3f(x1, y, z1);
        glVertex3f(x2, y, z2);
```

```
        glEnd();
    }
    glEnable(GL_LIGHTING);
    glPopMatrix();
}
```

绘制行星轨道时只需要将绘制号的轨道旋转行星的公转轨道倾角即可。

行星的轨道由轨道上的一些线段组成，整体效果为轨道上的虚线。

绘制行星轨道时，根据行星的轨道的长度计算出要绘制线段的数量（NUM）。

每次绘制时只需要计算出轨道上两点的坐标。

由于要绘制没有颜色的线段，所以在绘制时需要关掉环境光(glDisable(GL_LIGHTING))，

绘制完成后再打开。

5. 绘制环境光

```
void EnvironmentLight()
{
    // 光源 1
    GLfloat sun_light_position1[] = { 0.0f, 20000.0f * sizeRate, 0.0f,
1.0f };
    GLfloat sun_light_ambient1[] = { 0.2f, 0.2f, 0.2f, 1.0f };
    GLfloat sun_light_diffuse1[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    GLfloat sun_light_specular1[] = { 1.0f, 1.0f, 1.0f, 1.0f };

    glLightfv(GL_LIGHT0, GL_POSITION, sun_light_position1);
    glLightfv(GL_LIGHT0, GL_AMBIENT, sun_light_ambient1);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, sun_light_diffuse1);
    glLightfv(GL_LIGHT0, GL_SPECULAR, sun_light_specular1);


    // 光源 2
    GLfloat sun_light_position2[] = { 0.0f, -20000.0f * sizeRate, 0.0f,
1.0f };
    GLfloat sun_light_ambient2[] = { 0.2f, 0.2f, 0.2f, 1.0f };
    GLfloat sun_light_diffuse2[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    GLfloat sun_light_specular2[] = { 1.0f, 1.0f, 1.0f, 1.0f };
```

```
    glLightfv(GL_LIGHT1, GL_POSITION, sun_light_position2);
    glLightfv(GL_LIGHT1, GL_AMBIENT, sun_light_ambient2);
    glLightfv(GL_LIGHT1, GL_DIFFUSE, sun_light_diffuse2);
    glLightfv(GL_LIGHT1, GL_SPECULAR, sun_light_specular2);

    glEnable(GL_LIGHT0);
    //glEnable(GL_LIGHT1);
    glEnable(GL_LIGHTING);
    glEnable(GL_DEPTH_TEST);
    glDisable(GL_COLOR_MATERIAL);
}
```

将环境光设置为点光源，但是位置至于 y 轴的正方向上，效果就像在地图上方放置一个灯泡

6. 鼠标控制

```
void ActiveMotion(int x, int y)
{
    if (MouseX == 0 || MouseY == 0)
    {
        MouseX[MouseBuffer] = x;
        MouseY[MouseBuffer] = y;
        return;
    }

    Move[RIGHT] = x - MouseX[MouseBuffer] - 4;
    Move[LEFT] = x - MouseX[MouseBuffer] + 4;
    Move[UP] = y - MouseY[MouseBuffer] - 4;
    Move[DOWN] = y - MouseY[MouseBuffer] + 4;

    std::cout << x << " " << y << std::endl;

    MouseBuffer = (MouseBuffer + 1) % MOUSEBUFFERSIZE;

    MouseX[MouseBuffer] = x;
    MouseY[MouseBuffer] = y;
    SetCerma();
}

void SetCerma()
{
    if (Move[RIGHT] > 0)
    {
```

```
        eyeAngle1 += 2;
        while (eyeAngle1 >= 360)
            eyeAngle1 -= 360;
    }
    if (Move[LEFT] < 0)
    {
        eyeAngle1 -= 2;
        while (eyeAngle1 <= 360)
            eyeAngle1 += 360;
    }
    if (Move[UP] > 0)
    {
        if(eyeAngle2 <= 89)
            eyeAngle2 += 2;
    }
    if (Move[DOWN] < 0)
    {
        if(eyeAngle2 >= -89)
            eyeAngle2 -= 2;
    }
    GetCermaLocation();
}

void GetCermaLocation()
{
    eyeY = eyeR * sin(eyeAngle2 / 360 * 2 * M_PI);
    eyeZ = eyeR * cos(eyeAngle2 / 360 * 2 * M_PI) * cos(eyeAngle1 / 360
* 2 * M_PI);
    eyeX = eyeR * cos(eyeAngle2 / 360 * 2 * M_PI) * sin(eyeAngle1 / 360
* 2 * M_PI);
}
```

有一个全局变量的数组用来记录此时用户对鼠标的操控的状态。例如：如果检测出用户向右移动鼠标，则将 Move[RIGHT]置为正。如果用户向左移动鼠标，则将 Move[LEFT]的值置为负等。

由于 GLUT 提供的函数对鼠标的检测时每步都会进行的，但是用户在向上移动鼠标时可能会不小心左右波动鼠标，微小的波动应当忽略。故设置一鼠标缓冲区，每次检测鼠标之前 5 格的位置，如果用户的鼠标在之前位置的 4 个像素之内，则不视为用户移动了鼠标。

随后调用 GetCermaLocation 函数跟新相机的位置坐标。

效果：用户按下鼠标任意一个键同时移动鼠标，镜头会向用户移动的方向移动。

7. 键盘控制

```
void NormalKey(unsigned char key, int x, int y)
{
    switch (key)
    {
        case 'w':
        {
            if (eyeR >= 2500.0f * sizeRate)
                eyeR -= eyeR / 50;
        } break;
        case 's':
        {
            if (eyeR <= 80000.0f * sizeRate)
                eyeR += eyeR / 50;
        }break;
        case 'r':
        {
            Run = !Run;
        }break;
        default:
            break;
    }
    GetCermaLocation();
}

void SpecialKey(int key, int x, int y)
{
    switch (key)
    {
        case GLUT_KEY_LEFT:
            eyeAngle1 -= 2; break;
        case GLUT_KEY_RIGHT:
            eyeAngle1 += 2; break;
        case GLUT_KEY_UP:
            if(eyeAngle2 <= 89)
                eyeAngle2 += 2;
            break;
        case GLUT_KEY_DOWN:
            if(eyeAngle2 >= -89)
```

```
                eyeAngle2 -= 2;
            break;
        default:
            break;
    }

    GetCermaLocation();
}
```

OpenGL 将一些特殊键和能用 ASCII 码表示的键运用了不同的回调函数。

当用户按下普通键‘w’和‘s’，则代表拉近和拉远视角，每次拉近或拉远的距离为离世界中心位置的 2%。

用户按下上、下、左、右方向键时，视角会向上下左右移动。

按下‘r’键，则此时行星移动停止，再次按下则重新开始移动。

8. 数据设定

在资料中查到的太阳系的实际数据：

名称	英文名	半径 /km	自转周期/天	自转倾角/°	公转半径 /10^6km	公转周期/天	公转倾角/°
太阳	Sun	695500	25.05	7.25			
水星	Mercury	2439.7	58.646	2.11	57.9	115.88	7.005
金星	Venus	6051.8	-243	117.36	108.2	224.7	3.4
地球	Earth	6378	0.997	23.43	149.6	365.26	0
火星	Mars	3389.5	1.026	25.19	227.94	687	1.85
木星	Jupiter	69911	0.413	3.13	778.5	4332.59	1.305
土星	Saturn	60268	0.445	26.73	1443.45	10832.33	2.485
天王星	Uranus	25559	-0.718	97.77	2876.68	30799.1	0.77

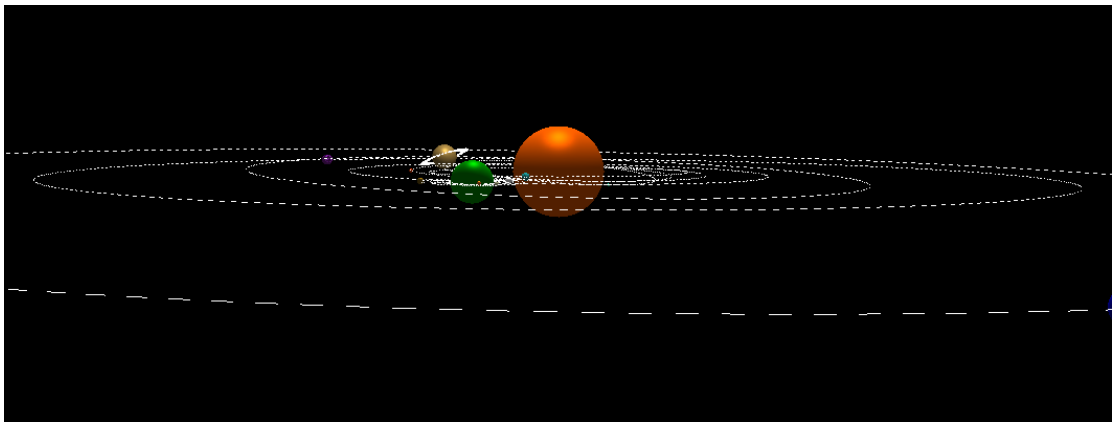
海王星	Neptune	24764	0.6	28.32	4503.44	60327.6	1.768
月球	Moon	1737.1	27.3		0.3844	27.3	5.145
木卫一	Lo	3660			0.42	1.77	0.05
木卫二	Europa	3121.6			0.67	3.55	0.47
木卫三	Ganymede	5262.4			1.07	7.15	0.204
木卫四	Calliso	4820.6			1.88	16.69	0.205

经过转换后实际绘制采用的数据：

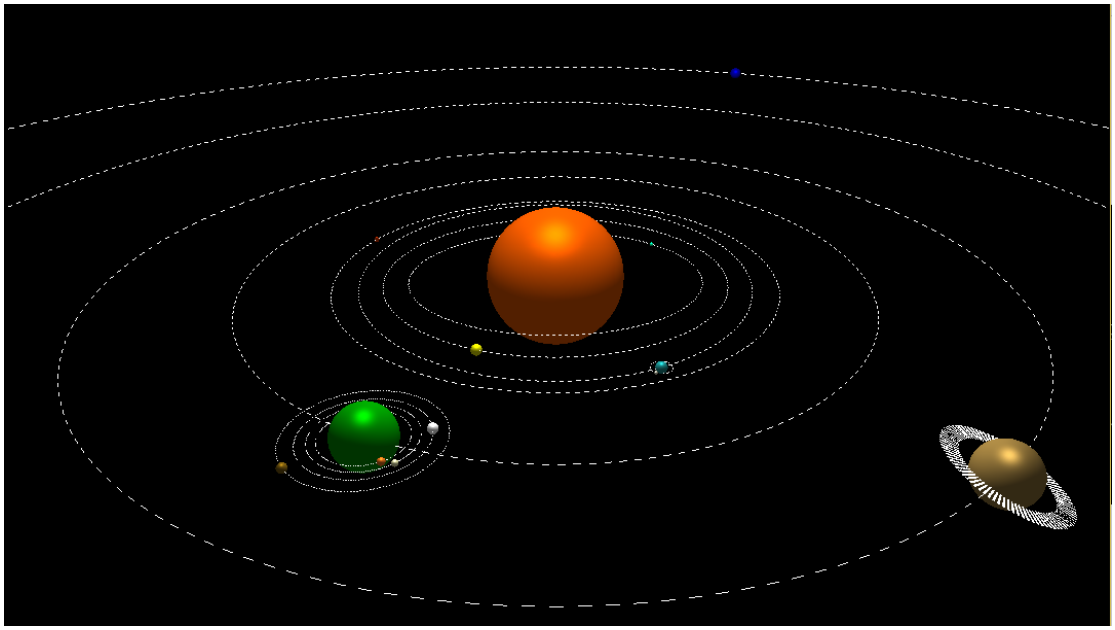
名称	**英文名**	**半径/单位**	**自转周期/单位**	**自转倾角/单位**	**公转半径/单位**	**公转周期/单位**	**公转倾角/单位**
太阳	Sun	2500	12.56				
水星	Mercury	58	29.32		4257	2.38	7.005
金星	Venus	146	-121.5		4980	4.62	3.4
地球	Earth	154	0.5		5664	7.5	0
火星	Mars	82	0.52		6412	14.1	1.85
木星	Jupiter	750	0.27		9000	58.96	1.305
土星	Saturn	690	0.23		13000	224.2	2.485
天王星	Uranus	309	-0.36		20000	632.41	0.77
海王星	Neptune	300	0.3		30000	1238.73	1.768
月球	Moon	41.94	13.65		350	0.63	5.145
木卫一	Lo	88.37			1000	1.77	0.05
木卫二	Europa	75.65			1100	3.55	0.47

木卫三	Ganymede	127.06		1300	7.15	0.204
木卫四	Calliso	116.39		1600	16.69	0.205

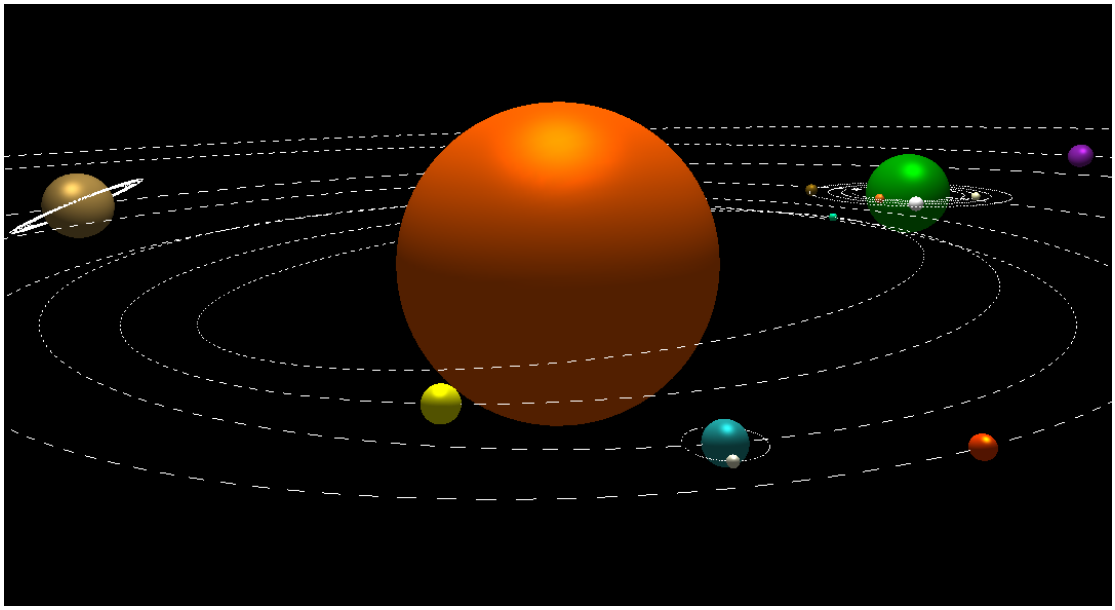
四、结果与分析



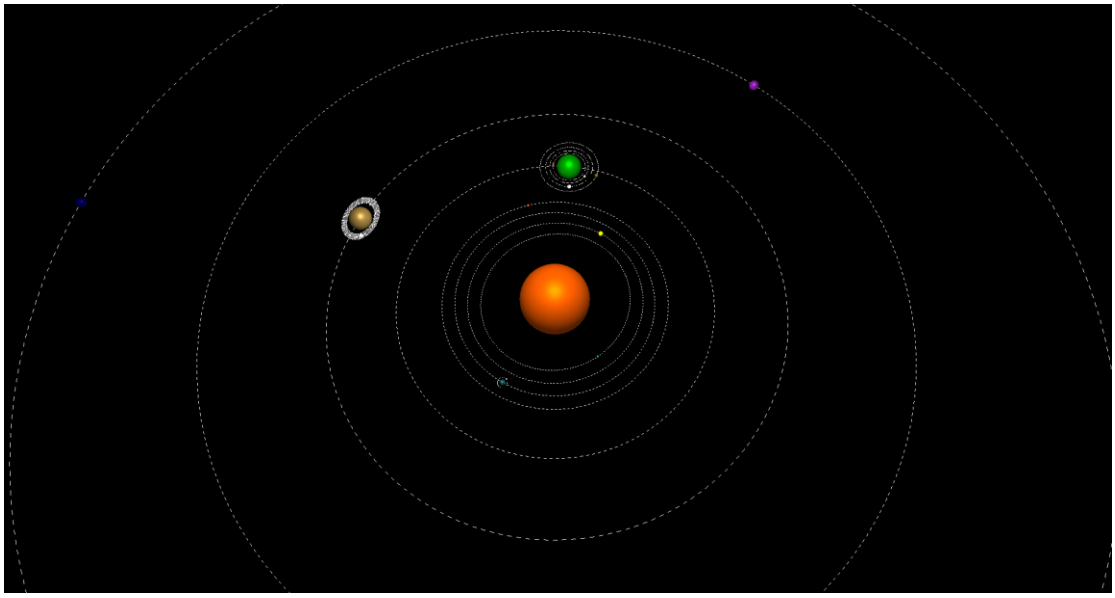
角度 1



角度 2



角度 3



角度 4