

C++ 实现太阳系行星系统 - 编码实现

一、说明

实验介绍

本节实验我们将对上一节实验中所设计的基本框架进行详细的实现。

实验涉及的知识点

- OpenGL 的矩阵模式
- OpenGL 常用的图像绘制接口
- OpenGL 中的视角
- OpenGL 的光照实现

二、基础知识

OpenGL 中的矩阵概念

在线性代数中，我们知道了矩阵的概念，但实际上我们对矩阵的具体作用和认识并没有体会到多少。矩阵到底是什么？

首先我们看这样一个式子：

$$x = Ab$$

其中 A 为矩阵， x, b 为向量。

观点一：

x 和 b 都是我们三维空间中的一个向量，那么 A 做了什么事情？—— A 把 b 变成了（变换到了） x 。在这样的观点下，矩阵 A 可以被理解为一种变换。

再来看另一个式子：

$$Ax = By$$

其中 A, B 为矩阵, x, y 为向量。

🔗 C++ 实现太阳系行星系统 (/courses/558)

观点二：

对于两个不同的向量 x, y 来说，它们本质上是同一个东西，因为它们只需要乘以矩阵 A, B 就可以相等。在这样的观点下，矩阵 A 可以被理解为一种坐标系。换句话说，向量本身是独一无二的，但是我们因为要描述它，所以定义了一个坐标系。由于坐标系取得不同，向量的坐标也会发生变化，即对于同一个向量来说，在不同的坐标系下有着不同的坐标。

矩阵 A 恰好描述了一个坐标系，而矩阵 B 也描述了一个坐标系，这两个坐标系作用到 x, y 上之后，得到了相同的结果，也即 x, y 本质是同一个向量，只不过他们有着不同的坐标系。

综合上述两个观点，所以矩阵的本质是：**描述运动**。

于是在 OpenGL 内部有一个负责绘制变换的矩阵，这就是 OpenGL 中的矩阵模式。

正如前面所说，矩阵既能够描述运动的变换，也能够描述某个物体所处的坐标系，因此在处理不同的操作时，我们要给 OpenGL 设置不同的矩阵模式，这就需要用到

```
glMatrixMode()
```

这个函数接受三个不同的模式：`GL_PROJECTION` 投影, `GL_MODELVIEW` 模型视图, `GL_TEXTURE` 纹理。

`GL_PROJECTION` 会向 OpenGL 声明将进行投影操作，会把物体投影到一个平面上。开启这个模式后要使用 `glLoadIdentity()` 把矩阵设置为单位矩阵，而后的操作如可以通过 `gluPerspective` 设置视景（在下面介绍完 OpenGL 中视角的概念后我们再详细说明这个函数的功能）。

`GL_MODELVIEW` 会向 OpenGL 声明接下来的语句将描绘一个以模型为基础的操作，比如设置摄像机视角，开启后同样需要设置 OpenGL 的矩阵模式为单位矩阵。

`GL_TEXTURE` 则是进行纹理相关的操作，我们暂时用不到。

如果对矩阵的概念不熟悉，可以直接将 `glMatrixMode` 理解为针对 OpenGL 申明接下来要做的事情。我们在绘制、旋转对象之前，一定要通过 `glPushMatrix` 保存当前的矩阵环境，否则会出现莫名其妙的绘制错误。

常用的 OpenGL 图像绘制 API

OpenGL 中提供了很多常用的与图形绘制相关的 API，这里我们挑几个常用的进行简单介绍，并在接下来的代码中进行使用：

- **`glEnable(GLenum cap)`**: 这个函数用于激活 OpenGL 中提供的各种功能，传入的参数 `cap` 是 OpenGL 内部的宏，提供诸如光源、雾化、抖动等效果；
- **`glPushMatrix()` 和 `glPopMatrix()`**: 将当前矩阵保存到堆栈栈顶（保存当前矩阵）；

- **glRotatef(alpha, x, y, z):** 表示当前图形沿 (x,y,z) 逆时针旋转 alpha 度;
- 🔗 C++ 实现太阳系行星系统 (/courses/558)
- **glTranslatef(distance, x, y):** 表示当前图形沿 (x,y) 方向平移 distance 距离;
- **glutSolidSphere(GLdouble radius, GLint slices, GLint stacks):** 绘制球体, radius 为半径, slices 为经线条数, stacks 为纬线条数;
- **glBegin()** 和 **glEnd()**: 当我们要进行图形绘制时, 需要在开始回之前和绘制完成后调用这两个函数, glBegin() 指定了绘制图形的类型, 例如 GL_POINTS 表示绘制点、GL_LINES 表示绘制依次画出的点及他们之间的连线、GL_TRIANGLES 则是在每绘制的三个点中完成一个三角形、GL_POLYGON 则是绘制一个从第一个点到第 n 个点的多边形, 等等。例如当我们需要绘制一个圆时可以边很多的多边形来模拟:

```
// r 是半径, n 是边数
glBegin(GL_POLYGON);
    for(i=0; i<n; ++i)
        glVertex2f(r*cos(2*PI/n*i), r*sin(2*PI/n*i));
glEnd();
```

OpenGL 里的视角坐标

上一节中, 我们在 SolarSystem 类中定义了九个成员变量

```
GLdouble viewX, viewY, viewZ;
GLdouble centerX, centerY, centerZ;
GLdouble upX, upY, upZ;
```

为了理解这九个变量, 我们首先需要树立起 OpenGL 三维编程中摄像机视角的概念。

想象平时我们观看电影的画面其实都是由摄像机所在的视角拍摄完成的, 因此 OpenGL 中也有类似的概念。如果我们把摄像机想象成我们自己的头, 那么:

1. viewX, viewY, viewZ 就相当于头 (摄像机) 在 OpenGL 世界坐标中的坐标位置;
2. centerX, centerY, centerZ 则相当于头所看 (摄像机所拍) 物体的坐标位置;
3. upX, upY, upZ 则相当于头顶 (摄像机顶部) 朝上的方向向量 (因为我们可以歪着头观察一个物体)。

至此, 你便有了 OpenGL 中坐标系的概念。

我们约定本次实验的初始视角在 (x, -x, x) 处, 则即有:

```
#define REST 700
#define REST_Y (-REST)
#define REST_Z (REST)
```

所观察物体（太阳）的位置在 (0,0,0)，则在 SolarSystem 类中的构造函数将视角初始化为：

🔗 C++ 实现太阳系行星系统 (/courses/558)

```
viewX = 0;
viewY = REST_Y;
viewZ = REST_Z;
centerX = centerY = centerZ = 0;
upX = upY = 0;
upZ = 1;
```

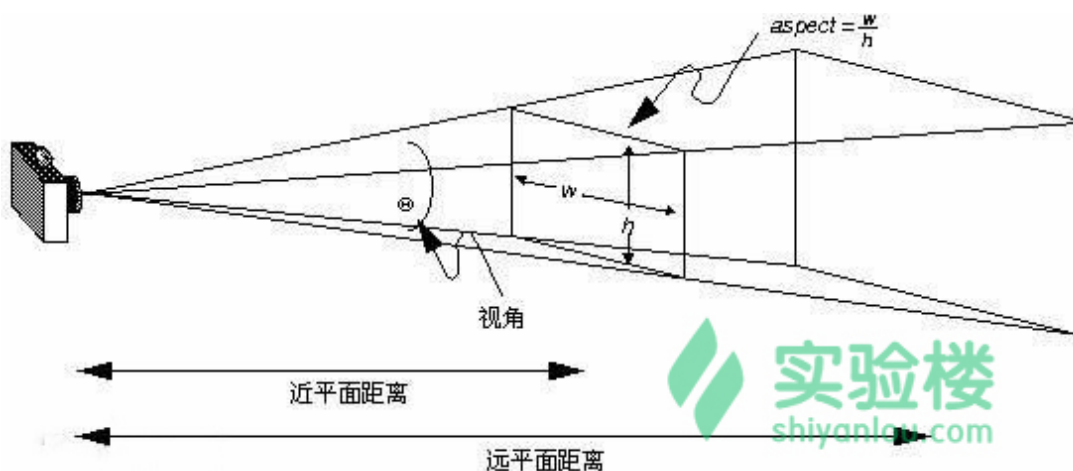
则可以通过 gluLookAt 函数来设置视角的九个参数：

```
gluLookAt(viewX, viewY, viewZ, centerX, centerY, centerZ, upX, upY, upZ);
```

然后我们再来看 gluPerspective(GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar)。

这个函数会创建一个对称的透视型视景体，在使用这个函数前需要将 OpenGL 的矩阵模式设置为 GL_PROJECTION。

如下图所示：



在窗口中的画面是通过摄像机来捕捉的，捕捉的实际内容就是远平面上的内容，显示的内容则是近平面上的内容，所以，这个函数需要四个参数：

1. 第一个参数为视角的大小
2. 第二个参数为实际窗口的横纵比，如图中 $aspect = w/h$
3. 第三个参数为近平面距离
4. 第四个参数则为远平面距离

OpenGL 里的光照效果

基本概念

OpenGL 在处理光照时将光照系统分为了三个部分：光源、材质、光照环境。

🔗 C++ 实现太阳系行星系统 (/courses/558)

顾名思义，光源就是光的来源，比如太阳；

材质这是指接受光照的各种物体的表面，比如太阳系中除太阳意外的行星和卫星都是这里所指的材质；

光照环境则是一些额外的参数，他们讲应将最终得到的光照面积，比如光线通常会经过多次反射，这时制定一个『环境亮度』的参数，可以使得最后的画面接近真实情况。

物理学里，平行光射入光滑平面上后所得的反射光依然是平行光，这种反射被叫做『镜面反射』；而对于不光滑的平面造成的反射，便就是所谓的『漫反射』。



光源

在 OpenGL 里要实现光照系统，我们首先需要做的就是设置光源。值得一提的是，OpenGL 内只支持有限数量的光源（八个）分别使用 `GL_LIGHT0` 至 `GL_LIGHT7` 这八个宏来表示。通过 `glEnable` 函数来启用，`glDisable` 函数来禁用。例如：`glEnable(GL_LIGHT0)`；

设置光源位置则需要使用 `glMaterialfv` 进行设置，例如：

```
GLfloat light_position[] = {0.0f, 0.0f, 0.0f, 1.0f};  
glLightfv(GL_LIGHT0, GL_POSITION, light_position); // 指定零号光源的位置
```

这里的位置由四个值来表示， (x, y, z, w) 其中当 w 为0时，表示该光源位于无限远，而 x, y, z 便指定了这个无限远光源的方向；

当 w 不为0时，表示位置性光源，其位置为 $(x/w, y/w, z/w)$ 。

材质

C++ 实现太阳系行星系统 (/courses/558)

设置一个物体的材质一般有五个属性需要设置：

1. 多次反射后追踪在环境中遗留的光照强度;
2. 漫反射后的光照强度;
3. 镜面反射后的光照强度;
4. OpenGL 中不发光物体发出的微弱且不影像其他物体的光照强度;
5. 镜面指数，指越小，表示材质越粗糙，点光源发射的光线照射后，产生较大的亮点；相反，值越大，材质越像镜面，产生较小的亮点。

设置材质 OpenGL 提供了两个版本的函数：

```
void glMaterialf(GLenum face, GLenum pname, TYPE param);
void glMaterialfv(GLenum face, GLenum pname, TYPE *param);
```

其差异在于，镜面指数只需要设置一个数值，这时只需要使用 `glMaterialf`；而其他的材质设置都需要设置多个值，这是需要使用数组进行设置，使用带指针向量参数的版本 `glMaterialfv`，例如：

```
GLfloat mat_ambient[] = {0.0f, 0.0f, 0.5f, 1.0f};
GLfloat mat_diffuse[] = {0.0f, 0.0f, 0.5f, 1.0f};
GLfloat mat_specular[] = {0.0f, 0.0f, 1.0f, 1.0f};
GLfloat mat_emission[] = {0.5f, 0.5f, 0.5f, 0.5f};
GLfloat mat_shininess = 90.0f;
glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
glMaterialf (GL_FRONT, GL_SHININESS, mat_shininess);
```

光照环境

OpenGL 是默认关闭光照处理的，要打开光照处理功能，需要使用 `GL_LIGHTING` 宏来激活，即 `glEnable(GL_LIGHTING)`；。

三、实现

行星的绘制

行星在绘制时，首先要考虑自身的公转角度和自转角度，因此我们可以先实现星球类的 `Star::update(long timeSpan)` 成员函数：

📁 C++ 实现太阳系行星系统 (courses/558)

```
alpha += timeSpan * speed; // 更新角度
alphaSelf += selfSpeed;    // 更新自转角度
}
```

在完成公转和自转角度的更新后，我们便可以根据参数绘制具体的星球了：

```
void Star::drawStar() {

    glEnable(GL_LINE_SMOOTH);
    glEnable(GL_BLEND);

    int n = 1440;

    // 保存 OpenGL 当前的矩阵环境
    glPushMatrix();
    {
        // 公转

        // 如果是行星，且距离不为0，那么 且向原点平移一个半径
        // 这部分用于处理卫星
        if (parentStar != 0 && parentStar->distance > 0) {
            //将绘制的图形沿 z 轴旋转 alpha
            glRotatef(parentStar->alpha, 0, 0, 1);
            // x 轴方向上平移 distance , y,z 方向不变
            glTranslatef(parentStar->distance, 0.0, 0.0);
        }
        // 绘制运行轨道
        glBegin(GL_LINES);
        for(int i=0; i<n; ++i)
            glVertex2f(distance * cos(2 * PI * i / n),
                       distance * sin(2 * PI * i / n));
        glEnd();
        // 绕 z 轴旋转 alpha
        glRotatef(alpha, 0, 0, 1);
        // x 轴方向平移 distance, y,z 方向不变
        glTranslatef(distance, 0.0, 0.0);

        // 自转
        glRotatef(alphaSelf, 0, 0, 1);

        // 绘制行星颜色
        glColor3f(rgbaColor[0], rgbaColor[1], rgbaColor[2]);
        glutSolidSphere(radius, 40, 32);
    }
    // 恢复绘制前的矩阵环境
    glPopMatrix();
}
```

这里用到了 `sin()` 和 `cos()` 函数，需要引入 `#include<cmath>`

光照的绘制

对于 Planet 类而言，属于不发光的星球，我们要绘制它的光照效果：

🔗 C++ 实现太阳系行星系统 (/courses/558)

```
void Planet::drawPlanet() {
    GLfloat mat_ambient[] = {0.0f, 0.0f, 0.5f, 1.0f};
    GLfloat mat_diffuse[] = {0.0f, 0.0f, 0.5f, 1.0f};
    GLfloat mat_specular[] = {0.0f, 0.0f, 1.0f, 1.0f};
    GLfloat mat_emission[] = {rgbaColor[0], rgbaColor[1], rgbaColor[2], rgbaColor[3]};
    GLfloat mat_shininess = 90.0f;

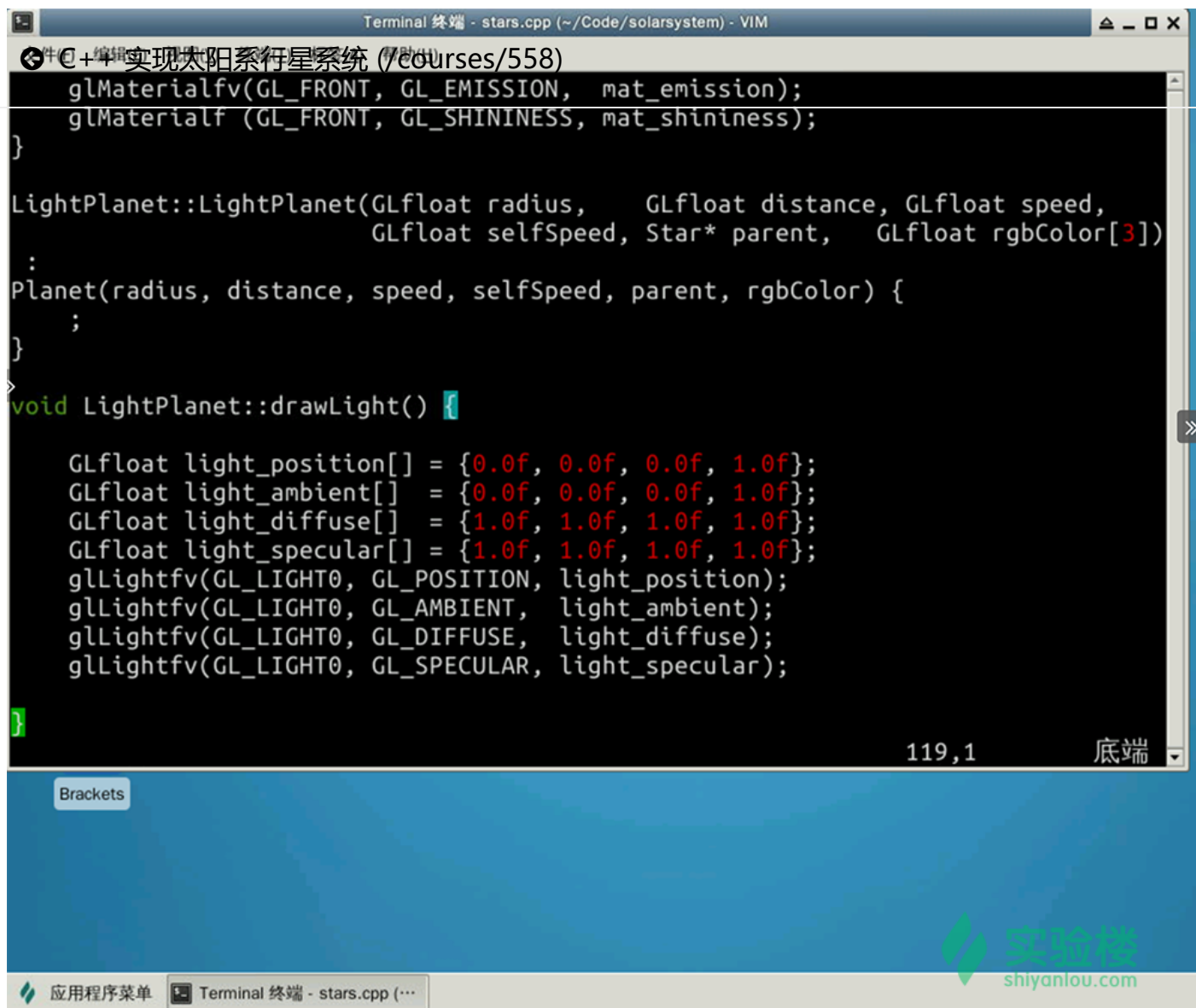
    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
    glMaterialf (GL_FRONT, GL_SHININESS, mat_shininess);
}
```

而对于 LightPlanet 类来说，属于发光的星球，所以我们不但要设置其光照材质，还要设置其光源位置：

```
void LightPlanet::drawLight() {

    GLfloat light_position[] = {0.0f, 0.0f, 0.0f, 1.0f};
    GLfloat light_ambient[] = {0.0f, 0.0f, 0.0f, 1.0f};
    GLfloat light_diffuse[] = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat light_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
    glLightfv(GL_LIGHT0, GL_POSITION, light_position); // 指定零号光源的位置
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient); // 表示各种光线照射到该材质上，经过很多次
    反射后追踪遗留在环境中的光线强度
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse); // 漫反射后的光照强度
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular); // 镜面反射后的光照强度

}
```

```
Terminal 终端 - stars.cpp (~ /Code/solarsystem) - VIM
C++ 实现太阳系行星系统 (/courses/558)
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
glMaterialf (GL_FRONT, GL_SHININESS, mat_shininess);
}

LightPlanet::LightPlanet(GLfloat radius,    GLfloat distance, GLfloat speed,
                        GLfloat selfSpeed, Star* parent,    GLfloat rgbColor[3])
:
Planet(radius, distance, speed, selfSpeed, parent, rgbColor) {
;
}

void LightPlanet::drawLight() {

    GLfloat light_position[] = {0.0f, 0.0f, 0.0f, 1.0f};
    GLfloat light_ambient[]  = {0.0f, 0.0f, 0.0f, 1.0f};
    GLfloat light_diffuse[]  = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat light_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightfv(GL_LIGHT0, GL_AMBIENT,  light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
}

119,1 底端
```

实现窗口的绘制

在上一节实验中我们提到过 `glutDisplayFunc` 、 `glutIdleFunc` 这两个处理图像显示的最重要的函数，`glutDisplayFunc` 会在 GLUT 确定窗口内容需要更新的时候将回调函数执行，`glutIdleFunc` 则是处理当事件循环空闲时的回调。

我们要实现整个太阳系运动起来，就应该考虑在什么时候更新行星的位置，在什么时候刷新视图。

显然，`glutDisplayFunc` 应该专注于负责刷新视图显示，当事件空闲时，我们便可以开始更新星球的位置，当位置更新完毕后，再调用视图刷新函数进行刷新。

因此，我们可以先实现 `glutDisplayFunc` 中调用的 `SolarSystem` 类中的成员函数 `SolarSystem::onUpdate()`：

 #define 实现太阳系行星系统 (courses/558)
void SolarSystem::onUpdate() {

```
    for (int i=0; i<STARS_NUM; i++)  
        stars[i]->update(TIMEPAST); // 更新星球的位置  
  
    this->onDisplay(); // 刷新显示  
}
```

其次，对于显示视图的刷新则是实现 SolarSystem::onDisplay()：

```
void SolarSystem::onDisplay() {  
  
    // 清除 viewport 缓冲区  
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);  
    // 清空并设置颜色缓存  
    glClearColor(.7f, .7f, .7f, .1f);  
    // 指定当前矩阵为投影矩阵  
    glMatrixMode(GL_PROJECTION);  
    // 将指定的矩阵指定为单位矩阵  
    glLoadIdentity();  
    // 指定当前的观察视景体  
    gluPerspective(75.0f, 1.0f, 1.0f, 40000000);  
    // 指定当前矩阵为视景矩阵堆栈应用术后的矩阵操作  
    glMatrixMode(GL_MODELVIEW);  
    // 指定当前的矩阵为单位矩阵  
    glLoadIdentity();  
    // 定义视图矩阵，并与当前矩阵相乘  
    gluLookAt(viewX, viewY, viewZ, centerX, centerY, centerZ, upX, upY, upZ);  
  
    // 设置第一个光源(0号光源)  
    glEnable(GL_LIGHT0);  
    // 启用光源  
    glEnable(GL_LIGHTING);  
    // 启用深度测试，根据坐标的远近自动隐藏被遮住的图形  
    glEnable(GL_DEPTH_TEST);  
  
    // 绘制星球  
    for (int i=0; i<STARS_NUM; i++)  
        stars[i]->draw();  
  
    // 我们在 main 函数中初始化显示模式时使用了 GLUT_DOUBLE  
    // 需要使用 glutSwapBuffers 在绘制结束后实现双缓冲的缓冲区交换  
    glutSwapBuffers();  
}
```

类的构造函数和析构函数

stars.hpp 中所定义类的构造函数需要对类中的成员变量进行初始化，相对较为简单，甚至可以使用默认析构，因此这部分请读者自行实现这些构造函数：

☞ C++ 实现太阳系行星系统 (courses/558)

```
Star::Star(GLfloat radius, GLfloat speed, GLfloat selfSpeed,
           Star* parent);
Planet::Planet(GLfloat radius, GLfloat distance,
               GLfloat speed, GLfloat selfSpeed,
               Star* parent, GLfloat rgbColor[3]);
LightPlanet::LightPlanet(GLfloat radius, GLfloat distance,
                          GLfloat speed, GLfloat selfSpeed,
                          Star* parent, GLfloat rgbColor[3]);
```

提示：注意在初始化速度变量时将其转化为角速度

其转换公式为： $\alpha_speed = 360/speed$

对于 solarsystem.cpp 的构造函数，我们需要对所有的星球进行初始化，这里为了方便起见我们先给出适当的行星之间的参数：

📁C4# 实现太阳系行星系统 (/courses/558)

```
#define SUN_RADIUS 48.74
#define MER_RADIUS 7.32
#define VEN_RADIUS 18.15
#define EAR_RADIUS 19.13
#define MOO_RADIUS 6.15
#define MAR_RADIUS 10.19
#define JUP_RADIUS 42.90
#define SAT_RADIUS 36.16
#define URA_RADIUS 25.56
#define NEP_RADIUS 24.78

// 距太阳的距离
#define MER_DIS 62.06
#define VEN_DIS 115.56
#define EAR_DIS 168.00
#define MOO_DIS 26.01
#define MAR_DIS 228.00
#define JUP_DIS 333.40
#define SAT_DIS 428.10
#define URA_DIS 848.00
#define NEP_DIS 949.10

// 运动速度
#define MER_SPEED 87.0
#define VEN_SPEED 225.0
#define EAR_SPEED 365.0
#define MOO_SPEED 30.0
#define MAR_SPEED 687.0
#define JUP_SPEED 1298.4
#define SAT_SPEED 3225.6
#define URA_SPEED 3066.4
#define NEP_SPEED 6014.8

// 自转速度
#define SELFROTATE 3

// 为了方便操作数组，定义一个设置多为数组的宏
#define SET_VALUE_3(name, value0, value1, value2) \
    ((name)[0])=(value0), ((name)[1])=(value1), ((name)[2])=(value2)

// 在上一节实验中我们定义了星球的枚举
enum STARS {Sun, Mercury, Venus, Earth, Moon,
    Mars, Jupiter, Saturn, Uranus, Neptune};
```

提示 C++ 实现太阳系行星系统 (/courses/558)

我们在这里定义了一个 `SET_VALUE_3` 的宏，读者或许会认为我们可以编写一个函数来达到快速设置的目的

事实上，宏会在编译过程就完成整体的替换工作，而定义函数

这需要在调用期间进行函数的堆栈操作，性能远不及编译过程就完成的宏处理

因此，使用宏会变得更加高效

但是值得注意的是，虽然宏能够变得更加高效，但过分的滥用则会造成代码的丑陋及弱可读性，而适当的使用宏则是可以提倡的

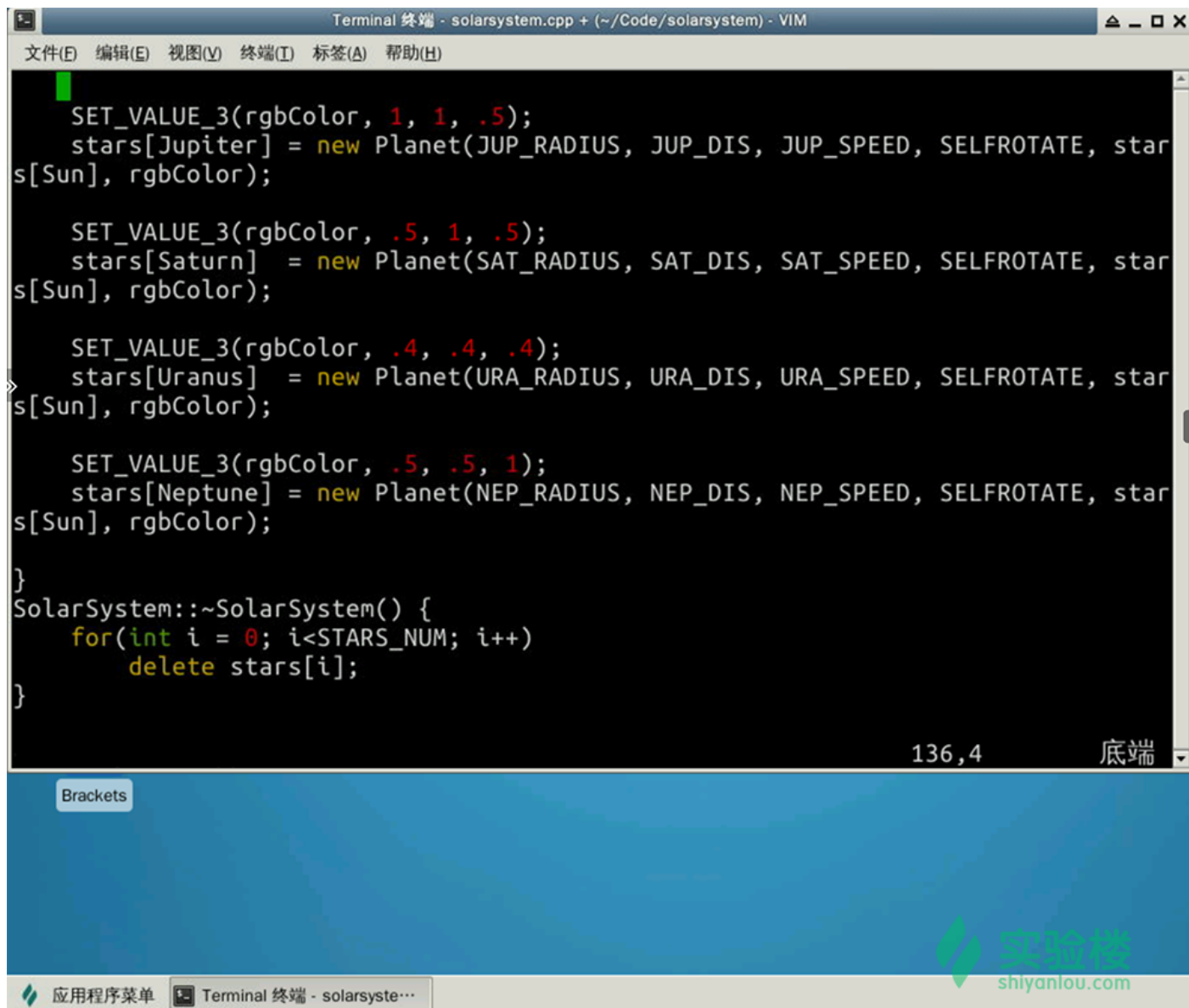
因此我们可以实现 `SolarSystem` 类的构造函数，其中星球的颜色是随机选取的，读者可以自行更改星球的颜色：

```
// 定义视角，在前面我们已经讨论过视角的初始化了
viewX = 0;
viewY = REST_Y;
viewZ = REST_Z;
centerX = centerY = centerZ = 0;
upX = upY = 0;
upZ = 1;

// 太阳
GLfloat rgbColor[3] = {1, 0, 0};
stars[Sun] = new LightPlanet(SUN_RADIUS, 0, 0, SELFROTATE, 0, rgbColor);
// 水星
SET_VALUE_3(rgbColor, .2, .2, .5);
stars[Mercury] = new Planet(MER_RADIUS, MER_DIS, MER_SPEED, SELFROTATE, stars[Sun], rgbColor);
// 金星
SET_VALUE_3(rgbColor, 1, .7, 0);
stars[Venus] = new Planet(VEN_RADIUS, VEN_DIS, VEN_SPEED, SELFROTATE, stars[Sun], rgbColor);
// 地球
SET_VALUE_3(rgbColor, 0, 1, 0);
stars[Earth] = new Planet(EAR_RADIUS, EAR_DIS, EAR_SPEED, SELFROTATE, stars[Sun], rgbColor);
// 月亮
SET_VALUE_3(rgbColor, 1, 1, 0);
stars[Moon] = new Planet(MOO_RADIUS, MOO_DIS, MOO_SPEED, SELFROTATE, stars[Earth], rgbColor);
// 火星
SET_VALUE_3(rgbColor, 1, .5, .5);
stars[Mars] = new Planet(MAR_RADIUS, MAR_DIS, MAR_SPEED, SELFROTATE, stars[Sun], rgbColor);
// 木星
SET_VALUE_3(rgbColor, 1, 1, .5);
stars[Jupiter] = new Planet(JUP_RADIUS, JUP_DIS, JUP_SPEED, SELFROTATE, stars[Sun], rgbColor);
// 土星
SET_VALUE_3(rgbColor, .5, 1, .5);
stars[Saturn] = new Planet(SAT_RADIUS, SAT_DIS, SAT_SPEED, SELFROTATE, stars[Sun], rgbColor);
// 天王星
SET_VALUE_3(rgbColor, .4, .4, .4);
stars[Uranus] = new Planet(URA_RADIUS, URA_DIS, URA_SPEED, SELFROTATE, stars[Sun], rgbColor);
// 海王星
SET_VALUE_3(rgbColor, .5, .5, 1);
stars[Neptune] = new Planet(NEP_RADIUS, NEP_DIS, NEP_SPEED, SELFROTATE, stars[Sun], rgbColor);
}
```

此外，不要忘了在析构函数中释放申请的内存：


```
SolarSystem::~SolarSystem() {  
    for(int i = 0; i<STARS_NUM; i++)  
        delete stars[i];  
}
```



```
Terminal 终端 - solarsystem.cpp + (~/.Code/solarsystem) - VIM  
文件(E) 编辑(E) 视图(V) 终端(T) 标签(A) 帮助(H)  
  
    SET_VALUE_3(rgbColor, 1, 1, .5);  
    stars[Jupiter] = new Planet(JUP_RADIUS, JUP_DIS, JUP_SPEED, SELFROTATE, stars[Sun], rgbColor);  
  
    SET_VALUE_3(rgbColor, .5, 1, .5);  
    stars[Saturn] = new Planet(SAT_RADIUS, SAT_DIS, SAT_SPEED, SELFROTATE, stars[Sun], rgbColor);  
  
    SET_VALUE_3(rgbColor, .4, .4, .4);  
    stars[Uranus] = new Planet(URA_RADIUS, URA_DIS, URA_SPEED, SELFROTATE, stars[Sun], rgbColor);  
  
    SET_VALUE_3(rgbColor, .5, .5, 1);  
    stars[Neptune] = new Planet(NEP_RADIUS, NEP_DIS, NEP_SPEED, SELFROTATE, stars[Sun], rgbColor);  
}  
SolarSystem::~SolarSystem() {  
    for(int i = 0; i<STARS_NUM; i++)  
        delete stars[i];  
}  
  
136,4 底端  
Brackets  
应用程序菜单 Terminal 终端 - solarsyste...  
shiyanlou.com
```

键盘按键变换视角的实现

我们不妨用键盘上的 `w,a,s,d,x` 五个键来控制视角的变换，并使用 `r` 键来复位视角。首先我们要确定一次按键后视角的变化大小，这里我们先定义一个宏 `OFFSET`。然后通过传入的 `key` 来判断用户的按键行为

 #define 实现太阳系行星系统 (/courses/558)

```
void SolarSystem::onKeyboard(unsigned char key, int x, int y) {
```

```
    switch (key)    {
        case 'w': viewY += OFFSET; break; // 摄像机Y 轴位置增加 OFFSET
        case 's': viewZ += OFFSET; break;
        case 'S': viewZ -= OFFSET; break;
        case 'a': viewX -= OFFSET; break;
        case 'd': viewX += OFFSET; break;
        case 'x': viewY -= OFFSET; break;
        case 'r':
            viewX = 0; viewY = REST_Y; viewZ = REST_Z;
            centerX = centerY = centerZ = 0;
            upX = upY = 0; upZ = 1;
            break;
        case 27: exit(0); break;
        default: break;
    }
}
```

四、总结本节中的代码(供参考)

本节实验中我们主要实现了 stars.cpp 和 solarsystem.cpp 这两个文件中的代码。

stars.cpp 的代码如下：

C++ 实现太阳系行星系统 (/courses/558)

```
// star.cpp
// solarsystem
//

#include "stars.hpp"
#include <cmath>

#define PI 3.1415926535

Star::Star(GLfloat radius, GLfloat distance,
           GLfloat speed, GLfloat selfSpeed,
           Star* parent) {
    this->radius = radius;
    this->selfSpeed = selfSpeed;
    this->alphaSelf = this->alpha = 0;
    this->distance = distance;

    for (int i = 0; i < 4; i++)
        this->rgbaColor[i] = 1.0f;

    this->parentStar = parent;
    if (speed > 0)
        this->speed = 360.0f / speed;
    else
        this->speed = 0.0f;
}

void Star::drawStar() {

    glEnable(GL_LINE_SMOOTH);
    glEnable(GL_BLEND);

    int n = 1440;

    glPushMatrix();
    {
        if (parentStar != 0 && parentStar->distance > 0) {
            glRotatef(parentStar->alpha, 0, 0, 1);
            glTranslatef(parentStar->distance, 0.0, 0.0);
        }
        glBegin(GL_LINES);
        for(int i=0; i<n; ++i)
            glVertex2f(distance * cos(2 * PI * i / n),
                      distance * sin(2 * PI * i / n));
        glEnd();
        glRotatef(alpha, 0, 0, 1);
        glTranslatef(distance, 0.0, 0.0);

        glRotatef(alphaSelf, 0, 0, 1);

        glColor3f(rgbaColor[0], rgbaColor[1], rgbaColor[2]);
        glutSolidSphere(radius, 40, 32);
    }
    glPopMatrix();
}
```

📁 C++ 实现太阳系行星系统 (courses/558)

```
alpha += timeSpan * speed;
alphaSelf += selfSpeed;
}

Planet::Planet(GLfloat radius, GLfloat distance,
               GLfloat speed, GLfloat selfSpeed,
               Star* parent, GLfloat rgbColor[3]) :
Star(radius, distance, speed, selfSpeed, parent) {
    rgbColor[0] = rgbColor[0];
    rgbColor[1] = rgbColor[1];
    rgbColor[2] = rgbColor[2];
    rgbColor[3] = 1.0f;
}

void Planet::drawPlanet() {
    GLfloat mat_ambient[] = {0.0f, 0.0f, 0.5f, 1.0f};
    GLfloat mat_diffuse[] = {0.0f, 0.0f, 0.5f, 1.0f};
    GLfloat mat_specular[] = {0.0f, 0.0f, 1.0f, 1.0f};
    GLfloat mat_emission[] = {rgbColor[0], rgbColor[1], rgbColor[2], rgbColor[3]};
    GLfloat mat_shininess = 90.0f;

    glMaterialfv(GL_FRONT, GL_AMBIENT, mat_ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
    glMaterialf (GL_FRONT, GL_SHININESS, mat_shininess);
}

LightPlanet::LightPlanet(GLfloat radius, GLfloat distance, GLfloat speed,
                          GLfloat selfSpeed, Star* parent, GLfloat rgbColor[3]) :
Planet(radius, distance, speed, selfSpeed, parent, rgbColor) {
    ;
}

void LightPlanet::drawLight() {

    GLfloat light_position[] = {0.0f, 0.0f, 0.0f, 1.0f};
    GLfloat light_ambient[] = {0.0f, 0.0f, 0.0f, 1.0f};
    GLfloat light_diffuse[] = {1.0f, 1.0f, 1.0f, 1.0f};
    GLfloat light_specular[] = {1.0f, 1.0f, 1.0f, 1.0f};
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
    glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);

}
```

solarsystem.cpp 的代码如下：

C++ 实现太阳系行星系统 (/courses/558)

```
// solarsystem.cpp
// solarsystem
//

#include "solarsystem.hpp"

#define REST 700
#define REST_Z (REST)
#define REST_Y (-REST)

void SolarSystem::onDisplay() {

    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glClearColor(.7f, .7f, .7f, .1f);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(75.0f, 1.0f, 1.0f, 40000000);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(viewX, viewY, viewZ, centerX, centerY, centerZ, upX, upY, upZ);

    glEnable(GL_LIGHT0);
    glEnable(GL_LIGHTING);
    glEnable(GL_DEPTH_TEST);

    for (int i=0; i<STARS_NUM; i++)
        stars[i]->draw();

    glutSwapBuffers();
}

#define TIMEPAST 1
void SolarSystem::onUpdate() {

    for (int i=0; i<STARS_NUM; i++)
        stars[i]->update(TIMEPAST);

    this->onDisplay();
}

#define OFFSET 20
void SolarSystem::onKeyboard(unsigned char key, int x, int y) {

    switch (key)    {
        case 'w': viewY += OFFSET; break;
        case 's': viewZ += OFFSET; break;
        case 'S': viewZ -= OFFSET; break;
        case 'a': viewX -= OFFSET; break;
        case 'd': viewX += OFFSET; break;
        case 'x': viewY -= OFFSET; break;
        case 'r':
            viewX = 0; viewY = REST_Y; viewZ = REST_Z;
            centerX = centerY = centerZ = 0;
            upX = upY = 0; upZ = 1;
            break;
        case 27: exit(0); break;
    }
```

```
default: break;
```

☛ C++ 实现太阳系行星系统 (/courses/558)

```
}

#define SUN_RADIUS 48.74
#define MER_RADIUS 7.32
#define VEN_RADIUS 18.15
#define EAR_RADIUS 19.13
#define MOO_RADIUS 6.15
#define MAR_RADIUS 10.19
#define JUP_RADIUS 42.90
#define SAT_RADIUS 36.16
#define URA_RADIUS 25.56
#define NEP_RADIUS 24.78

#define MER_DIS 62.06
#define VEN_DIS 115.56
#define EAR_DIS 168.00
#define MOO_DIS 26.01
#define MAR_DIS 228.00
#define JUP_DIS 333.40
#define SAT_DIS 428.10
#define URA_DIS 848.00
#define NEP_DIS 949.10

#define MER_SPEED 87.0
#define VEN_SPEED 225.0
#define EAR_SPEED 365.0
#define MOO_SPEED 30.0
#define MAR_SPEED 687.0
#define JUP_SPEED 1298.4
#define SAT_SPEED 3225.6
#define URA_SPEED 3066.4
#define NEP_SPEED 6014.8

#define SELFROTATE 3

enum STARS {Sun, Mercury, Venus, Earth, Moon,
            Mars, Jupiter, Saturn, Uranus, Neptune};

#define SET_VALUE_3(name, value0, value1, value2) \
            ((name)[0])=(value0), ((name)[1])=(value1), ((name)[2])=(value2)

SolarSystem::SolarSystem() {

    viewX = 0;
    viewY = REST_Y;
    viewZ = REST_Z;
    centerX = centerY = centerZ = 0;
    upX = upY = 0;
    upZ = 1;

    GLfloat rgbColor[3] = {1, 0, 0};
    stars[Sun] = new LightPlanet(SUN_RADIUS, 0, 0, SELFROTATE, 0, rgbColor);

    SET_VALUE_3(rgbColor, .2, .2, .5);
    stars[Mercury] = new Planet(MER_RADIUS, MER_DIS, MER_SPEED, SELFROTATE, stars[Sun], rgbCo
```

```
lor);
```

🔗 C++ 实现太阳系行星系统 (/courses/558)

```
SET_VALUE_3(rgbColor, 1, .7, 0);
```

```
stars[Venus] = new Planet(VEN_RADIUS, VEN_DIS, VEN_SPEED, SELFROTATE, stars[Sun], rgbColor);
```

```
SET_VALUE_3(rgbColor, 0, 1, 0);
```

```
stars[Earth] = new Planet(EAR_RADIUS, EAR_DIS, EAR_SPEED, SELFROTATE, stars[Sun], rgbColor);
```

```
SET_VALUE_3(rgbColor, 1, 1, 0);
```

```
stars[Moon] = new Planet(MOO_RADIUS, MOO_DIS, MOO_SPEED, SELFROTATE, stars[Earth], rgbColor);
```

```
SET_VALUE_3(rgbColor, 1, .5, .5);
```

```
stars[Mars] = new Planet(MAR_RADIUS, MAR_DIS, MAR_SPEED, SELFROTATE, stars[Sun], rgbColor);
```

```
SET_VALUE_3(rgbColor, 1, 1, .5);
```

```
stars[Jupiter] = new Planet(JUP_RADIUS, JUP_DIS, JUP_SPEED, SELFROTATE, stars[Sun], rgbColor);
```

```
SET_VALUE_3(rgbColor, .5, 1, .5);
```

```
stars[Saturn] = new Planet(SAT_RADIUS, SAT_DIS, SAT_SPEED, SELFROTATE, stars[Sun], rgbColor);
```

```
SET_VALUE_3(rgbColor, .4, .4, .4);
```

```
stars[Uranus] = new Planet(URA_RADIUS, URA_DIS, URA_SPEED, SELFROTATE, stars[Sun], rgbColor);
```

```
SET_VALUE_3(rgbColor, .5, .5, 1);
```

```
stars[Neptune] = new Planet(NEP_RADIUS, NEP_DIS, NEP_SPEED, SELFROTATE, stars[Sun], rgbColor);
```

```
}
```

```
SolarSystem::~~SolarSystem() {
```

```
    for(int i = 0; i<STARS_NUM; i++)
```

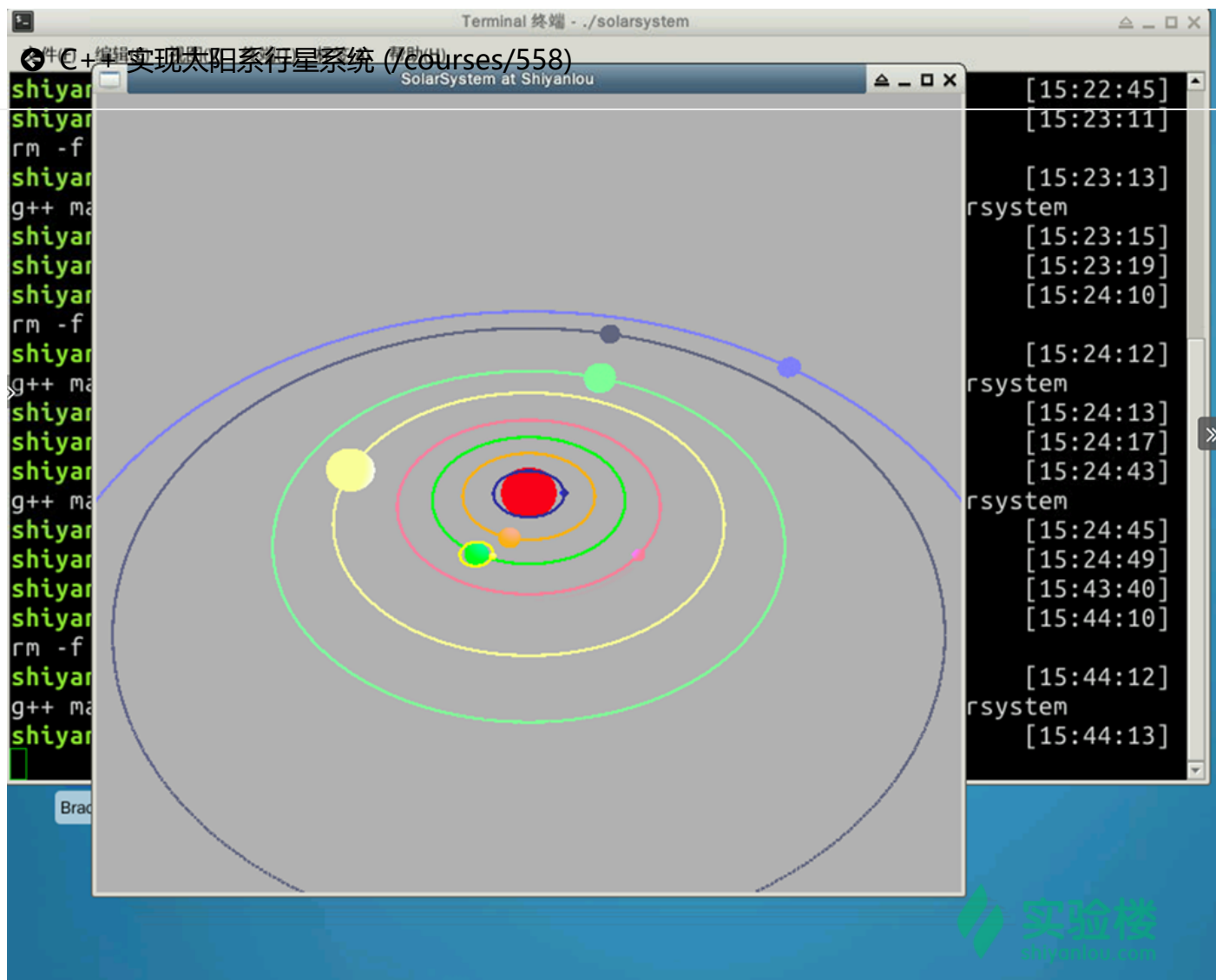
```
        delete stars[i];
```

```
}
```

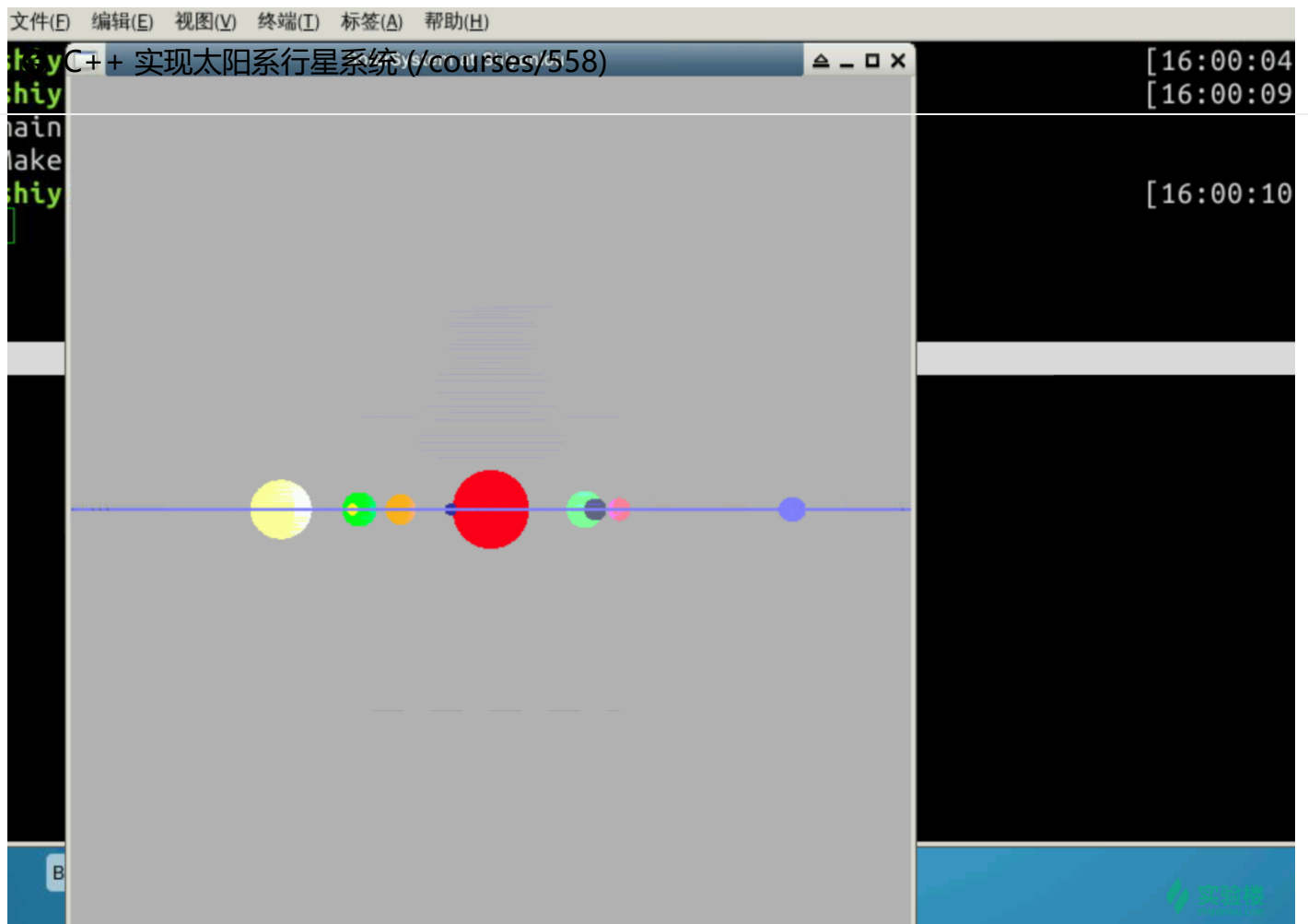
在终端中运行：

```
make && ./solarsystem
```

运行结果如图所示：



由于星球的颜色是单一的，光照效果不够明显但依然能够看到，诸如黄色的木星右侧可以看到有泛白，此外，我们还可以通过键盘来调整太阳系的查看视角：



本课程相关代码：

<http://labfile.oss.aliyuncs.com/courses/558/solarsystem.zip>

进一步阅读

1. D Shreiner. *OpenGL 编程指南*. 人民邮电出版社, 2005. 2.

这本书详细介绍了 OpenGL 编程相关的方方面面，被誉为『OpenGL 红宝书』

上一节：基本框架设计 (/courses/558/labs/1883/document)

