

基本框架设计

一、说明

实验介绍

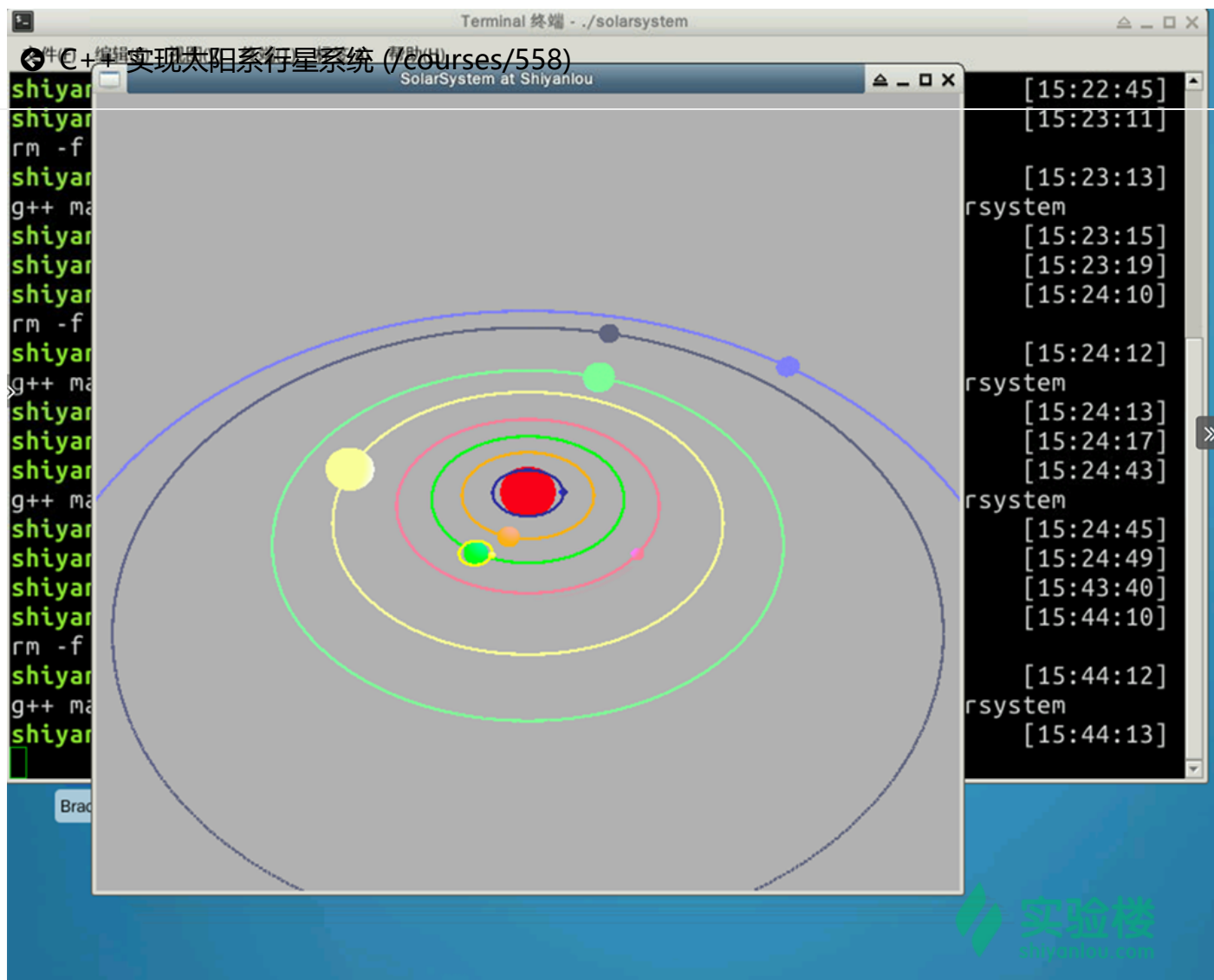
本次实验将使用 OpenGL GLUT 编写一个简单的太阳系运行系统。

实验涉及的知识点

- C++ 语言基础
- 基本的 Makefile
- 基本的 OOP 编程思想
- OpenGL GLUT 的结构基本使用

实验效果图

运行效果如图所示：



二、基础知识

认识 OpenGL 和 GLUT

OpenGL 包含了很多渲染函数，但是他们的设计目的是独立于任何窗口系统或操作系统的。因此，它自身并没有包含创建打开窗口或者从键盘或鼠标读取时间的函数，甚至连最基本的显示窗口的功能都没有，所以单纯只使用 OpenGL 是完全不可能创建一个完整的图形程序的。并且绝大多数程序都需要与用户进行交互（响应键盘鼠标等操作）。GLUT 则提供了这一便利。

GLUT 其实是 OpenGL Utility Toolkit 的缩写，它是一个处理 OpenGL 程序的工具库，主要负责处理与底层操作系统的调用及 I/O 操作。使用 GLUT 可以屏蔽掉底层操作系统 GUI 实现上的一些细节，仅使用 GLUT 的 API 即可跨平台的创建应用程序窗口、处理鼠标键盘事件等等。

我们先在实验楼环境中安装 GLUT：

```
sudo apt-get update && sudo apt-get install freeglut3 freeglut3-dev
```

一个标准的 GLUT 程序结构如下代码所示：

🔗 C++ 实现太阳系行星系统 (/courses/558)

// 使用 GLUT 的基本头文件

```
#include <GL/glut.h>
```

// 创建图形窗口的基本宏

```
#define WINDOW_X_POS 50
```

```
#define WINDOW_Y_POS 50
```

```
#define WIDTH 700
```

```
#define HEIGHT 700
```

// 用于注册 GLUT 的回调

```
void onDisplay(void);
```

```
void onUpdate(void);
```

```
void onKeyboard(unsigned char key, int x, int y);
```

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

```
    // 对 GLUT 进行初始化，并处理所有的命令行参数
```

```
    glutInit(&argc, argv);
```

```
    // 这个函数指定了使用 RGBA 模式韩式颜色索引模式。另外还可以
```

```
    // 指定是使用单缓冲还是双缓冲窗口。这里我们使用 RGBA 和 双缓冲窗口
```

```
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
```

```
    // 设置窗口被创建时左上角位于屏幕上的位置
```

```
    glutInitWindowPosition(WINDOW_X_POS, WINDOW_Y_POS);
```

```
    // 设置窗口被创建时的宽高，为了简便起见
```

```
    glutInitWindowSize(WIDTH, HEIGHT);
```

```
    // 创建一个窗口，输入的字符串为窗口的标题
```

```
    glutCreateWindow("SolarSystem at Shiyanlou");
```

```
    // glutDisplayFunc 的函数原型为 glutDisplayFunc(void (*func)(void))
```

```
    // 这是一个回调函数，每当 GLUT 确定一个窗口的内容需要更新显示的时候，
```

```
    // glutDisplayFunc 注册的回调函数就会被执行。
```

```
    //
```

```
    // glutIdleFunc(void (*func)(void)) 将指定一个函数，用于处理当事件循环
```

```
    // 处于空闲的时候，就执行这个函数。这个回调函数接受一个函数指针作为它的唯一参数
```

```
    //
```

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

```
    // 会将键盘上的键与一个函数关联，当这个键被按下或者释放时，函数就会调用
```

```
    //
```

```
    // 因此下面的三行实际上是在向 GLUT 注册关键的三个回调函数
```

```
    glutDisplayFunc(onDisplay);
```

```
    glutIdleFunc(onUpdate);
```

```
    glutKeyboardFunc(onKeyboard);
```

```
    glutMainLoop();
```

```
    return 0;
```

```
}
```

```
Terminal 终端 - main.cpp (~/.Code/solarsystem) - VIM
C++ 实验太阳系行星系统 (/courses/558)
#include "solarsystem.hpp"

#define WINDOW_X_POS 50
#define WINDOW_Y_POS 50
#define WIDTH 700
#define HEIGHT 700

SolarSystem solarsystem;


void onDisplay(void) {
    solarsystem.onDisplay();
}

void onUpdate(void) {
    solarsystem.onUpdate();
}

void onKeyboard(unsigned char key, int x, int y) {
    solarsystem.onKeyboard(key, x, y);
}

int main(int argc, char* argv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutInitWindowPosition(WINDOW_X_POS, WINDOW_Y_POS);
    glutCreateWindow("SolarSystem at Shiyanlou");
    glutDisplayFunc(onDisplay);
    glutIdleFunc(onUpdate);
    glutKeyboardFunc(onKeyboard);
    glutMainLoop();
    return 0;
}
```

31,1



提示

单缓冲，是将所有的绘图指令在窗口上执行，就是直接在窗口上绘图，这样的绘图效率是比较慢的，如果使用单缓冲，而电脑处理性能不够，屏幕会出现闪烁状。

双缓冲，会将绘图指令是在一个缓冲区完成，这里的绘图非常的快，在绘图指令完成之后，再通过交换指令把完成的图形立即显示在屏幕上，进而避免出现绘图的不完整，效率很高。

双缓冲则主要分为前台缓冲和后台缓冲，前台缓冲即我们说看到的屏幕，后台缓冲则维护内存中，对用户不可见。使用双缓冲时所有绘图操作都会在后台进行，当完成绘制后，才会将结果复制到屏幕上。

这么做的好处是，如果我们让绘制操作实时与显卡进行操作，当绘制任务复杂时，IO 操作同样会变得复杂，造成性能较低；而双缓冲只会在交换缓冲区时将绘制完成后的结果直接发送给显卡进行渲染，IO 显著降低。

在 OpenGL 中，推荐使用 GLfloat 来表示浮点数。

三、类设计

OOP 编程中首先要梳理清楚我们要处理的对象是什么。显然整个天体系统中，他们都是一颗星球 (Star)，区别行星和恒星只需要通过它们是否具有父节点即可；其次，对于不同的星球而言，它们通常具有自身的材质，并且不同的材质会表现出自身是否发光，由此我们有了初步的对象模型。故我们将星球分为：普通能够自转并绕某个点公转的星球(Star), 具有特殊材质的星球(Planet), 能够发光的星球(LightPlanet)

此外，为了编程实现上的方便，我们要对现实世界的实际编程模型做一些前提假设：

1. 星球的运行轨道为圆形；
2. 自转速度保持相同；
3. 每次刷新绘制的时候假设时间经过了一天。

首先，我们可以考虑按下面的思路整个实现逻辑：

1. 初始化星球对象;
2. 初始化 OpenGL 引擎, 实现 onDraw 和 onUpdate;
3. 星球应该自己负责处理自己的属性、绕行关系、变换相关绘制，因此在设计星球的类时应该提供一个绘制 draw() 方法;
4. 星球也应该自己处理自己自转公转等更新显示的绘制，因此在设计星球类时候也应该提供一个更新方法 update();
5. 在 onDraw() 中应调用星球的 draw() 方法;
6. 在 onUpdate() 中调用星球的 update() 方法;
7. 在 onKeyDown() 键盘调整整个太阳系的显示.

进一步，对于每个星球而言，都具有如下的属性：

1. 颜色 color
2. 公转半径 radius
3. 自转速度 selfSpeed
4. 公转速度 speed
5. 距离太阳中心的距离 distance
6. 绕行的星球 parentStar
7. 当前的自转的角度 alphaSelf
8. 当前的公转角度 alpha

根据前面的分析，我们可以设计如下的类代码：

📁 C++ 实现太阳系行星系统 (/courses/558)

```
public:
    // 星球的运行半径
    GLfloat radius;
    // 星球的公转、自传速度
    GLfloat speed, selfSpeed;
    // 星球的中心与父节点星球中心的距离
    GLfloat distance;
    // 星球的颜色
    GLfloat rgbColor[4];

    // 父节点星球
    Star* parentStar;

    // 构造函数，构造一颗星球时必须提供
    // 旋转半径、旋转速度、自转速度、绕行(父节点)星球
    Star(GLfloat radius, GLfloat distance,
        GLfloat speed, GLfloat selfSpeed,
        Star* parent);
    // 对一般的星球的移动、旋转等活动进行绘制
    void drawStar();
    // 提供默认实现，负责调用 drawStar()
    virtual void draw() { drawStar(); }
    // 参数为每次刷新画面时的时间跨度
    virtual void update(long timeSpan);
protected:
    Float alphaSelf, alpha;
};

class Planet : public Star {
public:
    // 构造函数
    Planet(GLfloat radius, GLfloat distance,
        GLfloat speed, GLfloat selfSpeed,
        Star* parent, GLfloat rgbColor[3]);
    // 增加对具备自身材质的行星绘制材质
    void drawPlanet();
    // 继续向其子类开放重写功能
    virtual void draw() { drawPlanet(); drawStar(); }
};

class LightPlanet : public Planet {
public:
    LightPlanet(GLfloat Radius, GLfloat Distance,
        GLfloat Speed, GLfloat SelfSpeed,
        Star* Parent, GLfloat rgbColor[]);
    // 增加对提供光源的恒星绘制光照
    void drawLight();
    virtual void draw() { drawLight(); drawPlanet(); drawStar(); }
};
```

```
Terminal 终端 - stars.hpp (~/.Code/solarsystem) - VIM
C++ 太阳系行星系统 7/courses/558)
Star(GLfloat radius, GLfloat distance,
    GLfloat speed, GLfloat selfSpeed,
    Star* parent);
void drawStar();
virtual void draw() { drawStar(); }
virtual void update(long timeSpan);
protected:
    GLfloat alphaSelf, alpha;
};

class Planet : public Star {
public:
    Planet(GLfloat radius, GLfloat distance,
        GLfloat speed, GLfloat selfSpeed,
        Star* parent, GLfloat rgbColor[3]);
    void drawPlanet();
    virtual void draw() { drawPlanet(); drawStar(); }
};

class LightPlanet : public Planet {
public:
    LightPlanet(GLfloat Radius, GLfloat Distance,
        GLfloat Speed, GLfloat SelfSpeed,
        Star* Parent, GLfloat rgbColor[]);
    void drawLight();
    virtual void draw() { drawLight(); drawPlanet(); drawStar(); }
};

#endif /* star_hpp */
```

44,0-1



此外，我们还需要考虑太阳系类的设计。在太阳系中，太阳系显然是由各个行星组成的；并且，对于太阳系而言，太阳系中行星运动后的视图刷新应该由太阳系来完成。据此太阳系成员变量应为包含行星的变量，成员函数应作为处理太阳系内的视图刷新及键盘响应等事件，所以，我们可以设计 Solar System 类：

```
class SolarSystem {
public:
    SolarSystem();
    ~SolarSystem();

    void onDisplay();
    void onUpdate();
    void onKeyboard(unsigned char key, int x, int y);

private:
    Star *stars[STARS_NUM];

    // 定义观察视角的参数
    GLdouble viewX, viewY, viewZ;
    GLdouble centerX, centerY, centerZ;
    GLdouble upX, upY, upZ;
};
```

提示 C++ 实现太阳系行星系统 (/courses/558)

1. 这里使用传统形式的数组来管理所有星球，而不是使用 C++ 中的 vector，是因为传统形式的数组就足够了
2. 在 OpenGL 中定义观察视角是一个较为复杂的概念，需要一定篇幅进行解释，我们先在此记下定义观察视角至少需要九个参数，我们将在下一节中具体实现时再详细讲解它们的作用。

最后我们还需要考虑一下基本的参数和变量设置。

在 SolarSystem 中，包括太阳在内一共有九颗星球（不包括冥王星），但是在我们所设计的 Star 类中，每一个 Star 对象都具有 parentStar 的属性，因此我们还可以额外实现这些星球的卫星，比如围绕地球运行的月球，据此我们一共考虑实现十个星球。于是我们可以设置如下枚举，用于索引一个数组中的星球：

```
#define STARS_NUM 10
enum STARS {
    Sun,           // 太阳
    Mercury,       // 水星
    Venus,         // 金星
    Earth,         // 地球
    Moon,          // 月球
    Mars,          // 火星
    Jupiter,       // 木星
    Saturn,        // 土星
    Uranus,        // 天王星
    Neptune       // 海王星
};
Star * stars[STARS_NUM];
```

我们还假设了自转速度相同，使用一个宏来设置其速度，：

```
#define TIMEPAST 1
#define SELFROTATE 3
```

至此，将未实现的成员函数创建到对应的 .cpp 文件中，我们便完成了本节实验。


```
Terminal 终端 - solarsystem.hpp (~/.Code/solarsystem) - VIM
C++ 实验太阳系行星系统 (/courses/558)

#ifndef solarsystem_hpp
#define solarsystem_hpp

#include <GL/glut.h>
#include "stars.hpp"

#define TIMEPAST 1
#define SELFROTATE 3

#define STARS_NUM 10
enum STARS {Sun, Mercury, Venus, Earth, Moon,
            Mars, Jupiter, Saturn, Uranus, Neptune};

class SolarSystem {
public:
    SolarSystem();
    ~SolarSystem();

    void onDisplay();
    void onUpdate();
    void onKeyboard(unsigned char key, int x, int y);
private:
    Star *stars[STARS_NUM];
};

#endif /* solarsystem_hpp */
```

34,1

实验楼
SHU 低端

总结本节中的代码

我们来总结一下本节实验中需要完成的代码：

首先我们在 `main.cpp` 中创建了一个 `SolarSystem`，然后将显示刷新、空闲刷新

及键盘事件的处理交给了 `glut`。

📁/C++ 实现太阳系行星系统 (/courses/558)

```
// main.cpp
// solarsystem
//
#include <GL/glut.h>
#include "solarsystem.hpp"

#define WINDOW_X_POS 50
#define WINDOW_Y_POS 50
#define WIDTH 700
#define HEIGHT 700

SolarSystem solarsystem;

void onDisplay(void) {
    solarsystem.onDisplay();
}
void onUpdate(void) {
    solarsystem.onUpdate();
}
void onKeyboard(unsigned char key, int x, int y) {
    solarsystem.onKeyboard(key, x, y);
}

int main(int argc, char* argv[]) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE);
    glutInitWindowPosition(WINDOW_X_POS, WINDOW_Y_POS);
    glutCreateWindow("SolarSystem at Shiyanlou");
    glutDisplayFunc(onDisplay);
    glutIdleFunc(onUpdate);
    glutKeyboardFunc(onKeyboard);
    glutMainLoop();
    return 0;
}
```

其次，我们创建了 Star/Planet/LightPlanet 类。

📁/C++ 实现太阳系行星系统 (/courses/558)

```
// stars.hpp
// solarsystem
//
//
#ifndef star_hpp
#define star_hpp

#include <GL/glut.h>

class Star {
public:
    GLfloat radius;
    GLfloat speed, selfSpeed;
    GLfloat distance;
    GLfloat rgbaColor[4];

    Star* parentStar;

    Star(GLfloat radius, GLfloat distance,
        GLfloat speed, GLfloat selfSpeed,
        Star* parent);
    void drawStar();
    virtual void draw() { drawStar(); }
    virtual void update(long timeSpan);
protected:
    GLfloat alphaSelf, alpha;
};

class Planet : public Star {
public:
    Planet(GLfloat radius, GLfloat distance,
        GLfloat speed, GLfloat selfSpeed,
        Star* parent, GLfloat rgbColor[3]);
    void drawPlanet();
    virtual void draw() { drawPlanet(); drawStar(); }
};

class LightPlanet : public Planet {
public:
    LightPlanet(GLfloat Radius, GLfloat Distance,
        GLfloat Speed, GLfloat SelfSpeed,
        Star* Parent, GLfloat rgbColor[]);
    void drawLight();
    virtual void draw() { drawLight(); drawPlanet(); drawStar(); }
};

#endif /* star_hpp */
```

stars.hpp 中对应类的成员函数实现:

📁/C++ 实现太阳系行星系统 (/courses/558)

```
// stars.cpp
// solarsystem
//
#include "stars.hpp"

#define PI 3.1415926535

Star::Star(GLfloat radius, GLfloat distance,
           GLfloat speed, GLfloat selfSpeed,
           Star* parent) {
    // TODO:
}

void Star::drawStar() {
    // TODO:
}

void Star::update(long timeSpan) {
    // TODO:
}

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

void Planet::drawPlanet() {
    // TODO:
}

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

void LightPlanet::drawLight() {
    // TODO:
}
```

设计了 SolarSystem 类：

📁/C++ 实现太阳系行星系统 (/courses/558)

```
// solarsystem.hpp
// solarsystem
//
#include <GL/glut.h>

#include "stars.hpp"

#define STARS_NUM 10

class SolarSystem {

public:

    SolarSystem();
    ~SolarSystem();

    void onDisplay();
    void onUpdate();
    void onKeyboard(unsigned char key, int x, int y);

private:
    Star *stars[STARS_NUM];

    // 定义观察视角的参数
    GLdouble viewX, viewY, viewZ;
    GLdouble centerX, centerY, centerZ;
    GLdouble upX, upY, upZ;
};
```

对应的 solarsystem.hpp 中的成员函数的实现:

☞/C++ 实现太阳系行星系统 (/courses/558)

```
// solarsystem.cpp
// solarsystem
//
#include "solarsystem.hpp"

#define TIMEPAST 1
#define SELFROTATE 3

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

void SolarSystem::onDisplay() {
    // TODO:
}
void SolarSystem::onUpdate() {
    // TODO:
}
void SolarSystem::onKeyboard(unsigned char key, int x, int y) {
    // TODO:
}
SolarSystem::SolarSystem() {
    // TODO:
}
SolarSystem::~SolarSystem() {
    // TODO:
}
```

Makefile 为

```
CXX = g++
EXEC = solarsystem
SOURCES = main.cpp stars.cpp solarsystem.cpp
OBJECTS = main.o stars.o solarsystem.o
LD_FLAGS = -lglut -lGL -lGLU

all :
    $(CXX) $(SOURCES) $(LD_FLAGS) -o $(EXEC)

clean:
    rm -f $(EXEC) *.gdb *.o
```



C++ 实现太阳系行星系统 (/courses/558)

书写编译命令时注意 `-lglut -lGLU -lGL` 的位置

这是因为 `g++` 编译器中 `-l` 选项用法有点特殊：

例如：`foo1.cpp -lz foo2.cpp`

如果目标文件 `foo2.cpp` 应用了库 `z` 中的函数，那么

这些函数不会被直接加载。而如果 `foo1.o` 使用了

`z` 库中的函数则不会出现任何编译错误。

换句话说，整个链接的过程是自左向右的。当 `foo1.cpp`

中遇到无法解析的函数符号时，会查找右边的链接库，当

发现选项 `z` 时，在 `z` 中查找然后发现函数，进而顺利完

成链接。

所以 `-l` 选项的库应该存在于所有编译文件的右边。

更多编译器细节请学习：[g++/gdb 使用技巧 \(\)](#)

进一步阅读

C++ 实现太阳系行星系统 (/courses/558)

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

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』。

下一节：编码实现 (/courses/558/labs/1884/document)