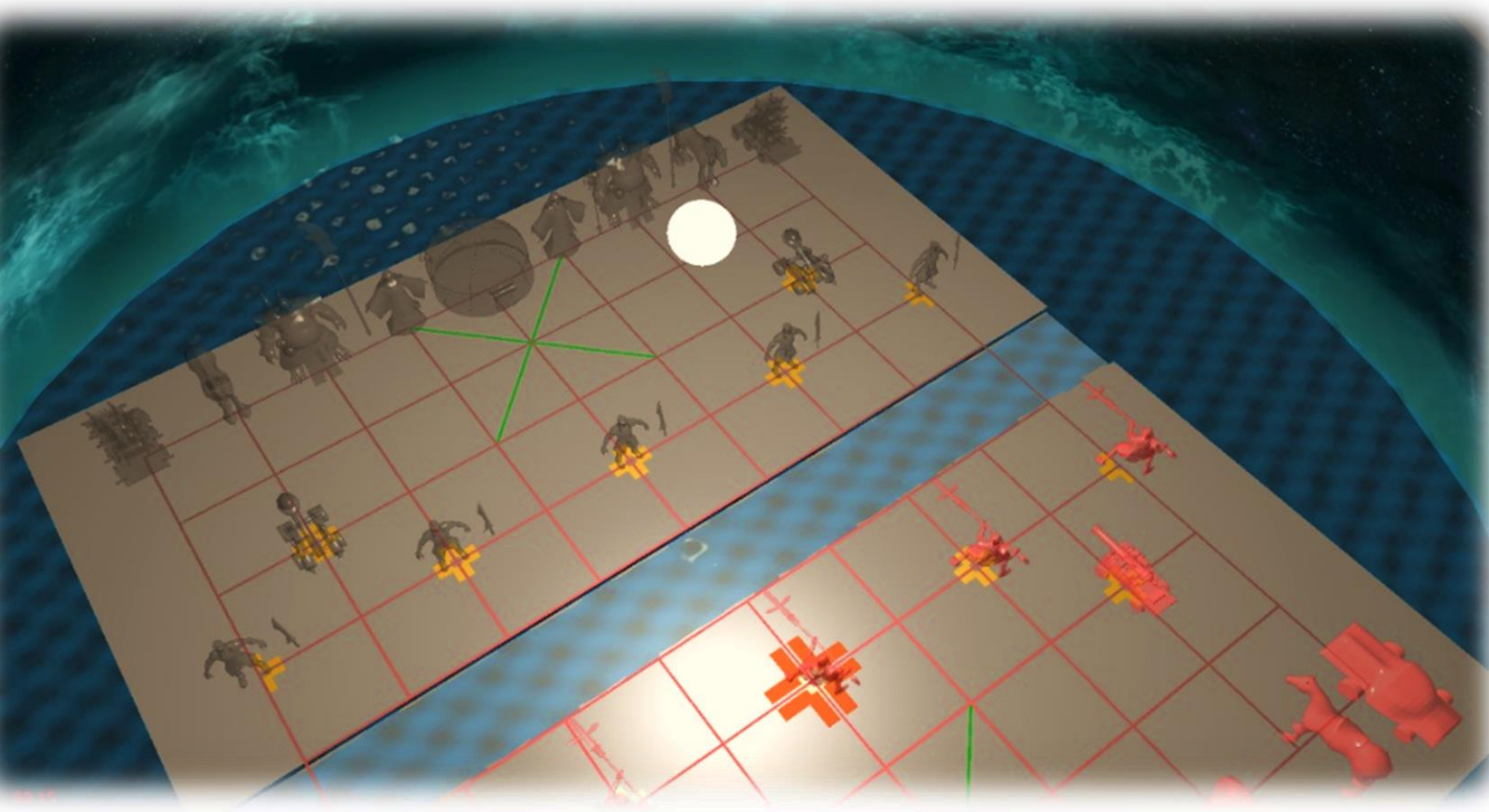
浙江大学

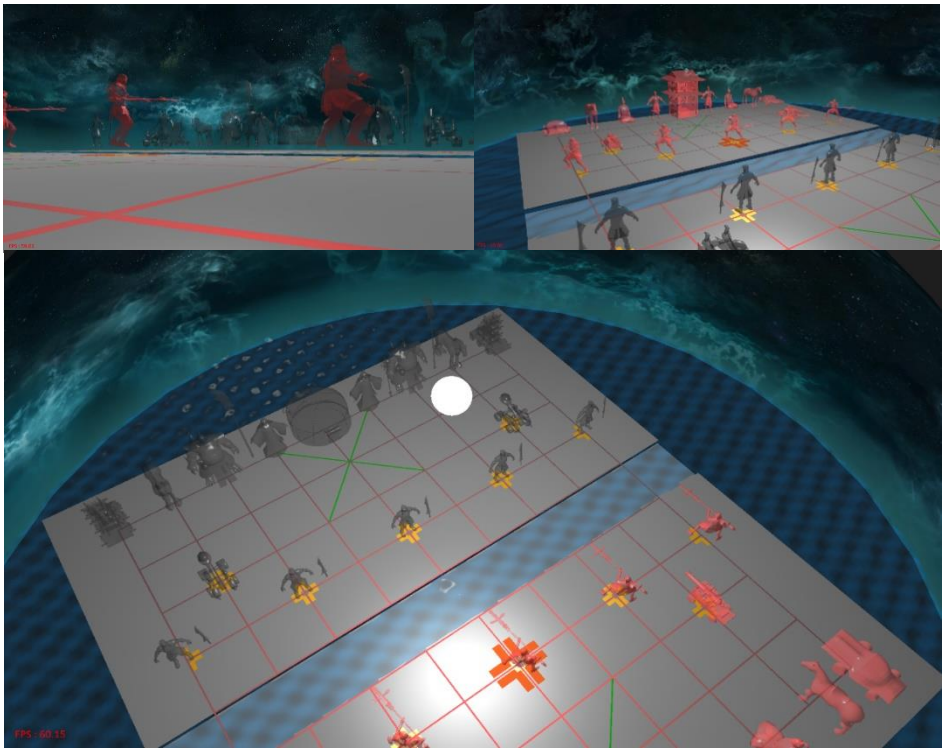
计算机图形学期末项目报告

基于 glfw/glad 的三维中国象棋游戏与场景建模



三维中国象棋

developed under glfw/glad



中国象棋是起源于中国的一种棋戏，属于二人对抗性游戏的一种，在中国有着悠久的历史。由于用具简单，趣味性强，成为流行极为广泛的棋艺活动。

中国象棋使用方形格状棋盘及红黑二色圆形棋子进行对弈，棋盘上有十条横线、九条竖线共分成 90 个交叉点；中国象棋的棋子共有 32 个，每种颜色 16 个棋子，分为 7 个兵种，摆放和活动在交叉点上。双方交替行棋，先把对方的将（帅）“将死”的一方获胜（因为擒贼先擒王）。已有几千年历史、充满东方智慧的中国象棋在中国的群众中基础远远超过围棋，一直是普及最广的棋类项目。

我们在 OpenGL 环境下，从头开始，搭建了适合我们渲染象棋场景的渲染引擎。并且基于此引擎，我们搭建了象棋场景，并且实现了一系列的类来管理象棋游戏的逻辑。

这份系统说明中，我们会对整个项目的设计框架和一些具体细节的实现做介绍。

第一章主要是对项目的整体结构做一个概述，介绍使用到的外部库，资源，着色器程序，以及 CPU 程序各模块的功能和定位。

接下来几章，我们挑选了一些重要的模块，详细介绍实现思路和细节。

浙江大学计算机图形学期末报告

指导老师：童若锋

小组成员：漆翔宇，张博康，张睿，代汶利

2019 年 1 月 2 日 星期三

目录

Chapter1 项目结构概述

项目文件组织

根目录下总共有六个子目录。
include： 存放头文件
source： 存放程序的.cpp/.c 文件
lib： 库目录
resource： 存放预设材质文件，棋子模型，天空球模型，字体等资源文件
shader： 存放着色器程序源文件
screenShot： 存放截图文件

外部库

GLFW/GLAD： 为了获得对渲染过程更加精细化的控制，本项目没有采用计算机图形学课程实验课上使用的固定渲染管线库。取而代之，我们选择了 LearnOpenGL CN 这个著名的 OpenGL 教程所推荐的一套支持核心渲染模式的 GLFW+GLAD 库。我们用这套库来初始化 OpenGL 上下文。

stb_Image： 我们使用它来加载各种格式的图片。所有的纹理图片，我们都是使用它载入到我们的程序中。

assimp： 用于加载模型文件的库，我们的所有.obj 格式的模型文件都是通过调用 assimp 提供的接口进行读取，分析，最后组织到我们程序的存储系统中来。

freetype： 用于生成文字贴图的库。我们的文字渲染，是通过载入 windows 目录下的标准字体文件，然后用这个库转换生成指定规格贴图，最后在指定位置，将带α分量的贴图贴在一块大小合适的透明矩形块上，渲染出来得到的。

glm： 一套数学库，提供了封装好的矩阵和向量类。并且提供了一系列方法，让我们对向量，矩阵做平移，旋转。由于核心渲染模式下，坐标变换都是在我们的控制下在顶点着色中完成的，所以三个变换矩阵都需要我们自己维护。glm 提供给我们的向量和矩阵对象有一套自带的函数来帮助我们进行变换计算。

在项目的 lib 目录下，你可以找到已经编译成库文件的 assimp/freetype/glfw 库，它们对应的头文件可以在 include 目录下的对应子目录下找到。glad/glm/stb_image 都是轻量级的辅助库，我们直接和项目一起编译。glad.c 可以在根目录下找到。stb_image 是一个只含头文件的图片加载程序，你可以在 include 目录下的 renderingEngine 中找到，glm 相关的文件你可以在 include 目下的 glm 目录下找到。

着色器程序

你可以在我们的 shader 目录下找到我们所有渲染过程中用到的着色器程序的代码。其中： objectFragmentShader.sd , objectVertexShader.sd , objectGeometryShader.sd 被用来渲染我们的模型文件。chessBoardVertexShader.sd , chessBoardFragmentShader.sd 被用来渲染棋盘。lightVertexShader.sd , lightFragmentShader.sd 被用来渲染光源。skyVertexShader.sd , skyFragmentShader.sd 用来渲染天空球。textFragmentShader.sd , textVertexShader.sd 用来渲染文本。waterFragmentShader.sd , waterVertexShader.sd 用来渲染水面。

源代码结构

不计入 glad.c，stb_image.cpp 这两个轻量级库的.c/.cpp 源文件，以及其他外部库所依赖的头文件，我们的项目总共有 18 个源文件。

项目是在 C++环境下开发的，但是采用了 Java 的风格。所有的类定义和类声明都写在了一起，没有分成.h 和.cpp。所以整个项目源文件除了 main 所在的 chineseChess.cpp 文件外全是.h 文件。

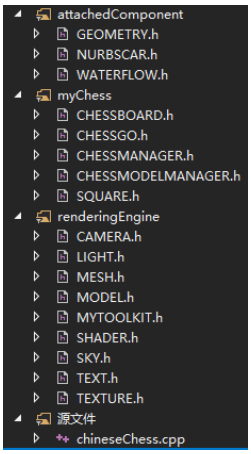


图 1 函数列表

（1）引擎组件

MYTOOLKIT.h

里面有一个类 Toolkit，这个类全是静态成员，不会被实例化，主要是用来做一个命名空间的封装。这个类的静态成员保存了我们当前使用的相机对象的指针，当前窗口的指针，当前窗口的大小。静态函数中提供了创造相机，创造着色器，初始化 OpenGL 上下文，截屏等接口。几乎所有的源文件都引入了这个头文件。像着色器，窗口，摄像机，光源等对象，都是全局共享的。Toolkit 类可以看作一个获得这些全局对象的接口。类似于一个记录上下文的笔记本。

SHADER.h

含一个 Shader 类。GLFW/GLAD 下，着色器的创建，链接和编译都是要手动完成的。我们把这套过程全部封装在一个 Shader 类中。这样只需要在实例化一个着色器程序的时候，把存放着色器代码文件的路径以字符串形式传入即可。所有的形式化的着色器编译，链接操作，以及相应的异常处理都被封装起来了。同时我们提供了更加简单的接口，让我们可以直接调用相应接口，把参数传到着色器程序的 uniform 变量处。

CAMERA.h

包含一个类 Camera，是相机类。成员变量包括鼠标敏感度，透视投影的参数，相机位置，相机指向方向，相机的上方指向的方向，透视投影矩阵和观察矩阵，以及相机可活动的范围，相机在垂直方向上的速度，是否出于自由模式等。

同时还含有几个静态成员变量记录重力加速度，相邻帧渲染耗时，当前窗口的指针，当前启用的相机的指针等。

其中静态方法 processInput 是用来检测键盘活动的。当前启用的相机的位置，方向，视角会受键盘活动影响，从而实现交互。

所有对相机位置进行改变的操作，都是通过成员方法 updatePos 完成的。这个方法会检测位置改变的请求是否合法。它会过滤掉不合法的移动操作。通过它，我们可以限制相机的移动范围，也可以进行漫游时的碰撞检测。

MODEL.h/MESH.h

分别含一个 Model 类和一个 Mesh 类

封装好的标准模型文件加载器，可以加载.obj 模型。MODEL.h 中含一个 Model 类，MESH.h 中含一个 Mesh 类。一个模型文件，由若干个网格组成，一个网格底层又是若干顶点。Model 对象调用 Assimp 提供的接口，读入模型文件，解析成网格，分别实例化成若干个 Mesh 对象，每个 Mesh 对象又进一步对网格数据解析，最后还原成顶点数据，然后将这些数据传到显存中，抽象成一个顶点数组对象。这样整个模型就转换成了我们可处理的数据格式。

实际使用中，我们只需要 Model 类这个顶层对象。实例化一个 Model 对象时，将模型文件的路径传入，整个模型就会自动化的被处理好。

TEXT.h

含一个 Text 类，是文本类，用来管理文本对象的渲染，在我们的项目中主要用来显示 FPS。

LIGHT.h

含一个类 Light，管理光源。在实例化一个光源的时候，把光源的类型，光色等属性通过

构造函数传入。所有设置片段着色器光源参数的操作都是通过它提供的 setLight 函数接口实现的，避免了大量的直接对着色器进行操作的代码的出现，全部封装到一个具体光源对象中实现。

同时，Light 对象本身也是可视化的。我们在点光源时，用一个球体来表示光源，这个球体会出现在设置点光源的位置，并且会呈现点光源的颜色。光源本身的模型和显示也是封装在 Light 对象中的。

SKY.h

含一个 sky 类,管理天空渲染,天空实际上就是一个带贴图的半球体。这个半球体是在建模软件中生成的，我们再把合适的贴图指定上去。导出为.obj 格式。所以一个 sky 对象就是接受一个天空球模型文件的路径，然后实例化一个 Model 对象来管理它，加载一个专用的 Shader 来渲染它。循环的画出这个巨大的天空球模型即可，然后就能营造出一种开阔感。

TEXTURE.h

含三个类。

MyTexture 类是管理贴图对象的，所有载入的贴图都被处理成 MyTexture 对象管理。天空球以及其他所有带贴图的模型，它们的贴图同样也是由这个类组织的。

Material 类管理材质,材质是直接和片段着色器挂钩的。我们的片段着色器中，有光源对象和材质对象。两个对象共同决定了最后的光照效果。我们这里提到的材质包括这个材质的漫反射颜色向量，环境光颜色向量，镜面反射光向量，以及反光度。我们预先参考了这个链接上的一些材质参数设置：http://devernay.free.fr/cours/opengl/materials.html 。把材质的名称和相应的参数都以文本形式存放在了/resource/material.txt 中，总计 24 种材质。Material 类中有一个静态函数，会把这些预设材质载入其一个类型为 vector 的静态容器中。

Name	Ambient			Diffuse			Specular			Shininess
emerald	0.0215	0.1745	0.0215	0.07568	0.61424	0.07568	0.633	0.727811	0.633	0.6
jade	0.135	0.2225	0.1575	0.54	0.89	0.63	0.316228	0.316228	0.316228	0.1
obsidian	0.05375	0.05	0.06625	0.18275	0.17	0.22525	0.332741	0.328634	0.346435	0.3
pearl	0.25	0.20725	0.20725	1	0.829	0.829	0.296648	0.296648	0.296648	0.088
ruby	0.1745	0.01175	0.01175	0.61424	0.04136	0.04136	0.727811	0.626959	0.626959	0.6
turquoise	0.1	0.18725	0.1745	0.396	0.74151	0.69102	0.297254	0.30829	0.306678	0.1
brass	0.329412	0.223529	0.027451	0.780392	0.568627	0.113725	0.992157	0.941176	0.807843	0.21794872
bronze	0.2125	0.1275	0.054	0.714	0.4284	0.18144	0.393548	0.271906	0.166721	0.2
chrome	0.25	0.25	0.25	0.4	0.4	0.4	0.774597	0.774597	0.774597	0.6
copper	0.19125	0.0735	0.0225	0.7038	0.27048	0.0828	0.256777	0.137622	0.086014	0.1
gold	0.24725	0.1995	0.0745	0.75164	0.60648	0.22648	0.628281	0.555802	0.366065	0.4
silver	0.19225	0.19225	0.19225	0.50754	0.50754	0.50754	0.508273	0.508273	0.508273	0.4
black plastic	0.0	0.0	0.0	0.01	0.01	0.01	0.50	0.50	0.50	.25
cyan plastic	0.0	0.1	0.06	0.0	0.50980392	0.50980392	0.50196078	0.50196078	0.50196078	.25
green plastic	0.0	0.0	0.0	0.1	0.35	0.1	0.45	0.55	0.45	.25
red plastic	0.0	0.0	0.0	0.5	0.0	0.0	0.7	0.6	0.6	.25
white plastic	0.0	0.0	0.0	0.55	0.55	0.55	0.70	0.70	0.70	.25
yellow plastic	0.0	0.0	0.0	0.5	0.5	0.0	0.60	0.60	0.50	.25
black rubber	0.02	0.02	0.02	0.01	0.01	0.01	0.4	0.4	0.4	.078125
cyan rubber	0.0	0.05	0.05	0.4	0.5	0.5	0.04	0.7	0.7	.078125
green rubber	0.0	0.05	0.0	0.4	0.5	0.4	0.04	0.7	0.04	.078125
red rubber	0.05	0.0	0.0	0.5	0.4	0.4	0.7	0.04	0.04	.078125
white rubber	0.05	0.05	0.05	0.5	0.5	0.5	0.7	0.7	0.7	.078125
yellow rubber	0.05	0.05	0.0	0.5	0.5	0.4	0.7	0.7	0.04	.078125

图 2 二十四种预设材质

Skybox 已经被弃用了，以前我们用天空盒来渲染环境，现在我们采用天空球了。但是我们还是保留了这个类。

（2）建模组件

NURBSCAR.h

含一个类:nurbsCar。管理我们用 nurbs 曲面建模建出来的车模型。我们的棋子模型中，红黑双方各 7 种，共 14 种模型。有 13 种是挑选的现成的.obj 模型。红方的车，我们是用 nurbs 曲面建模产生的。构造函数接受一个参数 n，表示曲面细分度，然后会从曲面上采样成 n*n 的网格，然后组织成顶点数组对象保存。

图 3　细分度为 50*50 的红方车和直接加载的黑方车模型

WATERFLOW.h

含一个类:waterFlow，管理我们的水面模型。构造器接受参数，设置水面的中心，纵横方向细分网格数，网格相邻的 gap。水面模型实际上只保存了 4 个顶点，即一个格子。整个水面网格采用的实例化方式绘制，每次渲染水面，直接并行的在 GPU 种绘制每一个网格就行了，同时水的波动计算都是在水面渲染专用的顶点着色器中进行的。充分利用了 GPU 的性能，节省了 CPU 的负担。

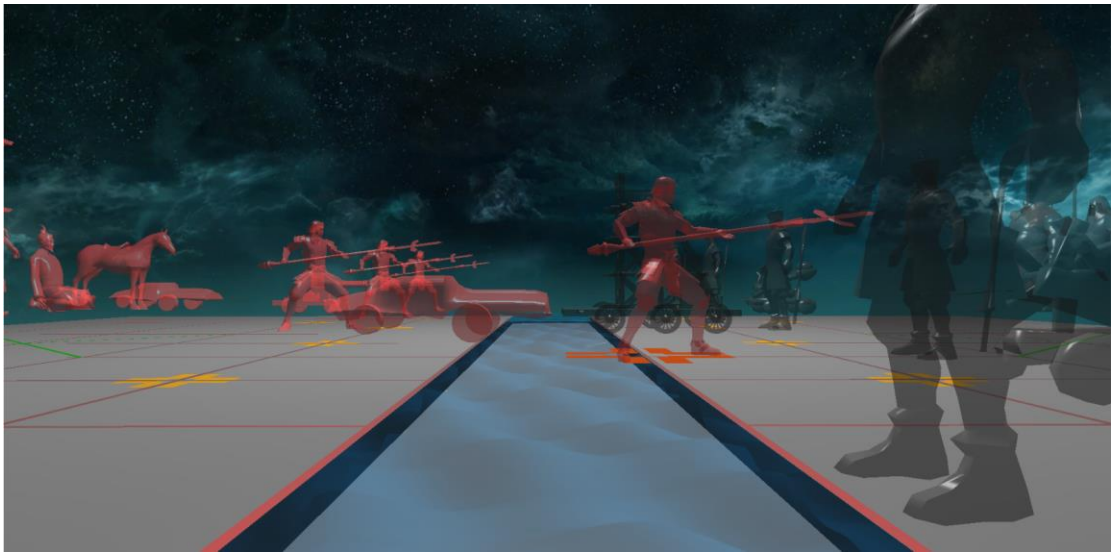


图 4　波动的“楚河”

GEOMETRY.h

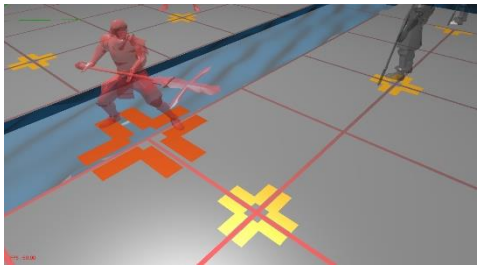
含多个类。其中 Geometry 是父类，其他的类都是它的子类。主要用来做基本几何体的建模。常见的基本几何体，如立方体，圆台，圆柱，球体，棱台等都做了建模封装。由于场景中需要的基本几何体不多，所以我们仅仅把楚河中心上方的点光源用一个球体表示了。但

所要求的几何体我们都做了封装。

（3）棋盘管理组件

SQUARE.h

图 5　棋盘材质展示



含一个类 Square,这个类的实例对象对应的就是一块矩形片。构造函数接受中心坐标，水平高度以及自身的宽度和长度。我们的棋盘盘面,棋盘上的线和标记,实际上底层都是它。

如右图，银质的盘面是一个大矩形块，实际上就是一个 Square 对象。金质的十字盘标，是由 8 块 Square 对象拼凑而成的，十字盘标被封装在另一个 Tag 类中，但是 Tag 类里实例化了 8 个 Square 对象。盘面上纵横交错的线条是红色橡胶材质，每一条线都是长窄的 Square 对象。

CHESSBOARD.h

含一个类 chessBoard，这是棋盘的顶层对象。这个类含有几个比较重要的成员。Square*Line[26]，是棋盘上纵横若干条线。boardCross *corss[2]，是棋盘上双方主将所行区域的那个×型线条。Square *groundObj[3];这是棋盘的盘面,我们切割成了三块进行渲染。chessManager* myManager 是用来管理棋盘逻辑的对象!!

这个顶层对象管理了棋盘盘身的渲染，同时麾下有一个棋盘管理器，它是 chessManager 类的实例化对象,这也是我们将要介绍的。

CHESSMANAGER.h

含四个类

boardCross 对应的是棋盘上 x 形线条。Tag 对应的是棋盘上兵，炮初始位置的几个 target 型图标。

Focus 对应的是我们棋盘上可移动的光标,这个类里面还封装了，关于 focus 事件的一些内容，它控制着光标移动，棋子选定等逻辑。

应该重点关注的是 chessManager 类,这是管理我们棋盘的类。可以看到它下面管理了 Focus 光标，水流面和一个棋子控制对象：chessGo* myChess。这个类的实例对象起到了管理盘面上可移动物的渲染和事件。

CHESSGO.h

含一个类 chessGo，它的实例化对象被用来具体的管理 32 个棋子。包括它们移动的合法性检测，游戏胜负的检测，棋子的移动动画控制，棋子被消灭的动画控制，以及棋子本身的布局逻辑。

麾下有一个 chessModelManager 类的对象。这个对象被用来具体的控制每一个棋子模型。chessGo 对象对每一个棋子的任何操作都是通过命令 chessModelManager 对象去完成底层实现的。

CHESSMODELMANAGER.h

含一个类 chessModel，主要是具体的存储了每个棋子的 Model 对象（或者 nurbsCar 对象），同时维护了每个棋子的模型矩阵。构造函数接受模型对象指针，放缩因子，旋转角度，平移量。这样可以让我们的模型在初始的时候，有一个合适的大小，朝向，并且对齐每一个落子点。

同时，chessModel 还提供了一个接口，接受一个平移量，对棋子模型进行平移。chessGo 对象就是调用的这一个方法，来实现棋子的运动。

（4）主程序

chineseChess.cpp :

主程序，调用 Toolkit 类中的相关方法，初始化 OpenGL 的上下文。

调用 Material 类的静态方法 initMaterail 加载我们预设的 24 种材质。

实例化 Light 类，创建光源对象。
实例化 Sky 类，创建天空球对象。
实例化着色器类，创建着色器对象。
将光源属性，更新到各个着色器程序中。
实例化文本类，初始化文本渲染器。
实例化相机类，创建相机，为相机设置移动范围。
实例化棋盘类，初始化整个棋盘模型。
然后主程序进入渲染循环，循环渲染天空，文本，光源，棋盘。计算每一帧的时差，更新到 Camera 类的静态成员中去。
计算帧率并通过文本渲染显示。

Chapter2 相机系统

在我们的摄像机中，我们建立了可以自由移动的摄像机，可以让玩家在棋盘中自由穿梭移动，可以来到模型底部一探究竟，也可以来到苍穹之上一览全局，甚是壮观。

而这些功能的实现离不开摄像机类的初始化，关于位置变化的一些函数，以及为了真实感我们设置的重力模式以及自由模式的切换控制。

关于摄像机移动的操作，我们简单做一下操作介绍：WASD 控制前后左右，空格上升，C 控制下降，F 切换重力与自由模式。

我们发现在摄像机自由移动的过程中，遇到模型会遇到穿模的情况，即穿过了模型，这显然是不合乎理性的。于是为了进一步增加真实感，我们引入了碰撞检测。

我们会在本章详细介绍摄像机类的初始化，关于改变摄像机位置、角度变化的函数，以及探讨如何切换重力模式与自由模式，最后讨论一下碰撞检测的实现。

摄像机类

关于我们的每一步操作，都有可能会改变摄像机的某些参数或者调用某些函数。所以我们用摄像机类将这一系列变量或函数封装起来。

```
class Camera
{
public:
    float sensitivity;
    float yaw, pitch, fov;
    glm::vec3 cameraPos;
    //接受来自 chessgo.h 信息
    glm::vec2 pos[9][10];
    int type[33];
    int locX[33];
    int locY[33];
    float sizeoftype[15][6];
    glm::vec3 cameraFront;
    glm::vec3 cameraUp;
    glm::mat4 projection;
    glm::mat4 view;
    float NearDis;
    float FarDis;
    float minX, minY, minZ, maxX, maxY, maxZ;
    float verticalSpeed;
    bool freeMode;
private:
    unsigned int width, height;
public:
    Camera(const unsigned int &width , const unsigned int &height)
    {
        yaw = -90.0f; pitch = 0; fov = 45.0f;
```

```
            cameraPos = glm::vec3(0.0f, 4.0f, 15.0f);
            cameraFront = glm::vec3(0.0f, 0.0f, -1.0f);
            cameraUp = glm::vec3(0.0f, 1.0f, 0.0f);
            NearDis = 0.1f;
            FarDis = 10000.f;
            projection = glm::perspective(glm::radians(45.0f), (float)width/
(float)height, NearDis, FarDis);
            this->width = width;
            this->height = height;
            sensitivity = 0.05;
            freeMode = true;
        }
        void updateMatrix(Shader * myShader,int type=0);
        float getSensitivity();
        void setSensitivity(float x);
        glm::mat4 getViewMatrix();
        glm::mat4 getProjectionMatrix();
        void setSpace(float cX, float cZ, float h, float wd, float lt, float maxH);
        void jump();
        void checkJump();
        void updatePos(glm::vec3 &temp);
        void changeMode();
        static Camera *currentCamera;
        static GLFWwindow *currentWindow;
        static float deltatime;
        static float lastFrame;
        static float gravity;
        static void mouse_callback(GLFWwindow* window, double xpos, double ypos);
        static void scroll_callback(GLFWwindow* window, double xoffset, double
yoffset);
        static void registerCamera(GLFWwindow *window,Camera *targetCamera);
        static void processInput();
    };
```

其中我们接收了来自 CHESSGO.h 文件中的棋子相关信息 pos[9][10]，type[33], locX[33], locY[33], sizeoftype[15][6],在之后的小节中我们将详细介绍是如何引用以及使用的。当然类中包含了关于摄像机的基本信息，cameraPos 用于存放当前摄像机的位置。cameraFront 表示摄像机朝向，cameraUp 为法向的朝向。minX, minY, minZ, maxX, maxY, maxZ 表示场景中的边框的坐标值，以免无限制地远离或者进入场景。

同时我们还定义了一些函数，改变与摄像机相关的静态变量，从而实现键盘对摄像机的控制。比如 void updatePos(glm::vec3 &temp)函数，static void processInput()函数。接下来我们详细介绍一下关于摄像机的变化。

摄像机位置、角度的变化

（1）摄像机位置的变化

在此部分中，我们通过调用 static void processInput()函数来处理键盘信息，并对摄像机的参数进行变化。

static glm::vec3 temp;我们定义了 temp，来储存摄像机将要到达的位置。

```
if (glfwGetKey(currentWindow, GLFW_KEY_W) == GLFW_PRESS)
        {
            temp = currentCamera->cameraPos + cameraSpeed *
currentCamera->cameraFront;
            currentCamera->updatePos(temp);
        }

        if (glfwGetKey(currentWindow, GLFW_KEY_S) == GLFW_PRESS)
        {
            temp = currentCamera->cameraPos - cameraSpeed *
currentCamera->cameraFront;
            currentCamera->updatePos(temp);
        }
        if (glfwGetKey(currentWindow, GLFW_KEY_A) == GLFW_PRESS)
        {
            temp = currentCamera->cameraPos - cameraSpeed *
glm::normalize(glm::cross(currentCamera->cameraFront, currentCamera->cameraUp));
            currentCamera->updatePos(temp);
        }
        if (glfwGetKey(currentWindow, GLFW_KEY_D) == GLFW_PRESS)
        {
            temp=currentCamera->cameraPos + cameraSpeed *
glm::normalize(glm::cross(currentCamera->cameraFront, currentCamera->cameraUp));
            currentCamera->updatePos(temp);
        }
        if (glfwGetKey(currentWindow, GLFW_KEY_SPACE) == GLFW_PRESS)
        {
            if (currentCamera->freeMode)
            {
                temp = currentCamera->cameraPos + cameraSpeed *
currentCamera->cameraUp;
                currentCamera->updatePos(temp);
            }
            else
            {
                if (spaceDown == false)
                {
                    currentCamera->jump();
                    spaceDown = true;
                }
            }
        }
        if (glfwGetKey(currentWindow, GLFW_KEY_SPACE) ==
```

```
GLFW_RELEASE)spaceDown = false;
        if (glfwGetKey(currentWindow, GLFW_KEY_C) == GLFW_PRESS)
        {
            temp=currentCamera->cameraPos - cameraSpeed *
currentCamera->cameraUp;
            currentCamera-> updatePos (temp);
            }
```

这是一些关于摄像机移动的键盘回调信息处理，根据相应的键我们对 temp 向量分别进行变化，然后调用 updatePos 函数来判断。在 updatePos 函数中，我们将会判断该移动是否合法，若合法，则间接访问 cameraPos 变量进行改变，若不合法，则不改变 cameraPos 向量，摄像机位置就不会更新。

（2）摄像机角度的变化：

```
static void mouse_callback(GLFWwindow* window, double xpos, double ypos)
{
    static bool firstMouse=true;
    static double lastX = 0;
    static double lastY = 0;
    if (firstMouse)
    {
        lastX = xpos;
        lastY = ypos;
        firstMouse = false;
        return;
    }
    float xoffset = xpos - lastX;
    float yoffset = lastY - ypos;
    lastX = xpos;
    lastY = ypos;

    xoffset *= currentCamera->sensitivity;
    yoffset *= currentCamera->sensitivity;

    currentCamera->yaw += xoffset;
    currentCamera->pitch += yoffset;

    if (currentCamera->pitch > 89.0f)currentCamera->pitch = 89.0f;
    if (currentCamera->pitch < -89.0f)currentCamera->pitch = -89.0f;

    glm::vec3 front;
    front.x = cos(glm::radians(currentCamera->yaw)) *
cos(glm::radians(currentCamera->pitch));
    front.y = sin(glm::radians(currentCamera->pitch));
    front.z = sin(glm::radians(currentCamera->yaw)) *
cos(glm::radians(currentCamera->pitch));
```

```
        currentCamera-> cameraFront = glm::normalize(front);
    }
```

通过调用 mouse_callback 函数，检测鼠标的变化，计算朝向坐标进而改变 cameraFront 变量从而改变摄像机的朝向。

重力模式与自由模式

在重力模式中，我们的摄像机始终有一个高度坐标向下的 分量，空格只能做跳跃动作，即无法离开表面太多，给了游戏更多的真实感和体验。

fDown 变量判断摄像机此时是否处于下降状态，freeMode 标记是否为重力模式。

```
void changeMode()
    {
        freeMode = !freeMode;
    }
```

changeMode 函数可以改变模式类型。我们调用此函数改变 freeMode 变量。

在 processInput()函数中，我们会判断键盘信息以及当前状态是否处于重力模式，或者是否改变当前状态，最后调用 checkJump()函数。

`temp.y = cameraPos.y + verticalSpeed * (deltatime);`来改变摄像机的高度，接下来进行碰撞检测。

碰撞检测

之前的三个小节或多或少都涉及到了碰撞检测，接下来我们细致讨论一下碰撞检测。因为我们已经有了棋子的逻辑，当棋子碰撞到了棋子，那么在象棋的规则中，该棋子即替代了之前在此位置的棋子，也就是原来的棋子被吃掉了，这个在之后的章节中我们会再详细讨论如何替换的以及替换过程的渲染。

首先，我们将碰撞检测的步骤分为三步。

第一步：我们要将每个棋子设定碰撞范围。

因为我们的棋子模型的实现都非常精确，我们采取较为可行的方法实现检测，即将每个棋子都设置在一个碰撞范围内，我们此次将每个棋子都放入了一个立方体盒子中。显然每个棋子的盒子大小各不相同，我们的“将”“帅”模型体积较大，“士”体积较小，也就是说我们要为各个棋子做一个合适的盒子。我们通过观察具体渲染以及调用模型的头文件，设置了 14 种盒子，因为总共红蓝两方共有 14 种棋子。

在 CHESSGO.h 中，我们初始化 sizeoftype[15][6]数组，来记录各类型棋子盒子的大小：

```
    float sizeoftype[15][6] = {//分别为x,z,y相应大小
        0,0,0,0,0,0,
        -10,16,-4,4,4,22,
        -10,12,-5,5,4,12,
        -14,10.8,-8,6.8,4,11,
        -11,14,-3.6,3.6,5,20,
        -6.2,6.7,-5.7,5.7,7,21.5,
        -3.5,3,-8.5,8.5,5,24,
        -20,13.6,-16.8,16.8,4,37,
```

```
    -1.8,2.4,-6,12,4,20,
    -11,9.1,-5.45,5.45,4,15,
    -10.18,10.9,-7.27,7.27,5.8,22,
    -12.57,14.3,-2.85,2.85,5.0,22.5,
    -6.67,8.33,-12.2,13.8,6.4,28.5,
    -5.23,5.71,-8.09,11.90,5.5,27.2,
    -14.05,11.89,-12.97,12.97,5.7,21.6
};
```

第二步：我们要在 CAMERA.h 中的 updatePos 函数中进行判断，检测与每个盒子检测相交。事实上，我们的盒子是一个虚拟的概念，只是在碰撞检测的过程中，为每个模型设置了一个在碰撞时的检测标准，即在该棋子本来的世界坐标上根据此参数设置一个盒子，然后用摄像机的世界坐标去检测是否相撞。

首先我们遇到的第一个问题，我们在 CAMERA.h 头文件中进行摄像机与棋子的碰撞检测，但是我们在此头文件中并没有各棋子的任何坐标信息，于是我们采用 myToolkit::myCamera 中设置一些变量来记录关于棋子的坐标信息，然后在 CHESSGO.h 中实时更新这些信息，我们就可以在 CAMERA.h 中使用这些信息来进行判断了，我们用同样的方法引用 CHESSGO.h 的关于棋子盒子的信息。

在 CAMERA.h 头文件中初始化相关信息，接收来自 CHESSGO.h 的信息：

```
glm::vec2 pos[9][10];
int type[33];
int locX[33];
int locY[33];
float sizeoftype[15][6];
```

在 CHESSGO.h 中，在 initialBoard()初始化棋盘的函数中，进行传送信息:

```
for (int i = 0; i < 33; i++)
    {
        Toolkit::myCamera->locX[i] = locX[i];
        Toolkit::myCamera->locY[i] = locY[i];
        Toolkit::myCamera->type[i] = type[i];

    }
    for (int i = 0; i < 9; i++)
    {
        for (int j = 0; j < 10; j++)
        {
            Toolkit::myCamera->pos[i][j] = pos[i][j];
        }
    }
    for (i = 0; i < 15; i++)
    {
        for (j = 0; j < 6; j++)
        {
```

```
                Toolkit::myCamera->sizeoftype[i][j] = sizeoftype[i][j];
            }
        }
```

现在我们有了棋子的各类坐标信息，需要说明的是，通过 locX[id],locY[id]我们可以获得这个棋子在棋盘矩阵上的行号和列号，通过 pos[rowId][colId]可以获得棋盘矩阵对应位置的世界坐标。接下来我们便可以进行判断了，在 updatePos 函数中：

```
for (int i = 0; i < 33; i++)
        {
            //判断各自棋子类型的大小 sizeoftype[][]存储了该类棋子的大小
            if (type[i] > 0)//判断是否该棋子还在棋盘中
            {
                if (pos[locX[i]][locY[i]].x + sizeoftype[type[i]][0] < temp.x &&
temp.x < pos[locX[i]][locY[i]].x + sizeoftype[type[i]][1])
                {
                    if (pos[locX[i]][locY[i]].y + sizeoftype[type[i]][2] < temp.z &&
    temp.z < pos[locX[i]][locY[i]].y + sizeoftype[type[i]][3])
                    {
                        {
                            if (sizeoftype[type[i]][4] < temp.y&&temp.y <
sizeoftype[type[i]][5])
                            {

                                return;
                            }
                        }
                    }
                }
            }

        }
```

即我们判断摄像机的下一步移动是否进入某个还在棋盘上的象棋的“盒子”中，如果进入，显然是不允许的，所以直接 return，不对摄像机的坐标进行更新，如果没有进入，那么也不一定是合法的，我们还要进行一步碰撞检测，即与场景周边的碰撞检测：

```
if (temp.x <= maxX && temp.x >= minX)cameraPos.x = temp.x;
        else if (temp.x < minX)cameraPos.x = minX;
        else if (temp.x > maxX)cameraPos.x = maxX;

        if (temp.y <= maxY && temp.y >= minY)cameraPos.y = temp.y;
        else if (temp.y < minY)cameraPos.y = minY;
        else if (temp.y > maxY)cameraPos.y = maxY;

        if (temp.z <= maxZ && temp.z >= minZ)cameraPos.z = temp.z;
```

```
        else if (temp.z < minZ)cameraPos.z = minZ;
        else if (temp.z > maxZ) cameraPos.z = maxZ;

        return;
```

只有经过这两步检测，即既不与棋子相撞，又不与场景边框相撞，才能更新摄像机坐标，进行移动。

讨论到这里，不要忘记刚才在重力模式的时候，我们曾经设置了摄像机高度坐标的改变，所以我们在 checkJump()函数中同样进行了碰撞检测。

话已至此，我们好像已经做好了碰撞检测。但是其实我们忽略了一点：我们现在完成了移动的摄像机向棋子移动的碰撞检测，显然，我们还应该要进行移动的棋子对摄像机的碰撞检测，而这种检测则分为两种情况：

一种情况是摄像机的位置在棋子行进之后所处的位置。因为 CHESSGO.h 中进行了棋子坐标的移动，而摄像机位置的变化我们可以通过 myToolkit：：myCamera 间接访问来改变，所以我们直接在 CHESSGO.h 中进行判断，即判断摄像机的世界坐标是否在棋子将要到达位置的盒子内，如果在，那么摄像机就要被弹开，我们假设碰撞过程遵循弹性碰撞并且假设质量相等，那么摄像机就要被弹开一定距离，该距离等于碰撞之前的距离。

判断碰撞代码如下：

```
if (pos[locX[id]][locY[id]].x + sizeoftype[type[id]][0] <Toolkit::myCamera->cameraPos.x
    &&  Toolkit::myCamera->cameraPos.x < pos[locX[id]][locY[id]].x +
sizeoftype[type[id]][1])
        {
            if (pos[locX[id]][locY[id]].y + sizeoftype[type[id]][2]
<Toolkit::myCamera->cameraPos.z&& Toolkit::myCamera->cameraPos.z <
pos[locX[id]][locY[id]].y + sizeoftype[type[id]][3])
            {
                {
                    if (sizeoftype[type[id]][4] < Toolkit::myCamera->cameraPos.y&&
    Toolkit::myCamera->cameraPos.y < sizeoftype[type[id]][5])
                    {
                        flag = 1;//flag=1，棋子与摄像机相撞
                        cout << "what" << "\n";
                    }

                }
            }
        }
```

若碰撞，我们将被撞开的位移分解，因为如果撞到了其他棋子或者到达边界同样要停下。部分位置改变代码如下：

```
if (flag == 1)
        {
            for (int i = 0; i < 100; i++)
            {
                temp.x = temp.x + tinyx;
                temp.z = temp.z + tinyy;
```

```
                flag = 0;
                for (int j = 0; j < 33; j++)//除了该棋子其他的判断
                {
                    if (j == id)
                        continue;
                    //判断各自棋子类型的大小 sizeoftype[][]存储了该类棋子的大小
                    if (type[j] >= 0)//判断是否该棋子还在棋盘中
                    {
                        if (pos[locX[j]][locY[j]].x + sizeoftype[type[j]][0] < temp.x
    &&  temp.x < pos[locX[j]][locY[j]].x + sizeoftype[type[j]][1])
                        {
                            if (pos[locX[j]][locY[j]].y + sizeoftype[type[j]][2] <
temp.z  && temp.z < pos[locX[j]][locY[j]].y + sizeoftype[type[j]][3])
                            {
                                {
                                    if (sizeoftype[type[j]][4] < temp.y&&temp.y <
sizeoftype[type[j]][5])
                                        flag = 1;
                                }
                            }
                        }
                    }

                }
```

这是一种棋子主动撞向摄像机的一种情况，那么还有另外一种。棋子的行进过程中撞开了摄像机（之前一种是棋子的到达位置撞开摄像机）。那么我们要判断一下是否在行进过程中撞到，此时又要分为两种情况，是同列变换还是同行变换撞到的，不同的碰撞方式进行不同的位移，我们预定摄像机被一直撞到棋子的位置之后一步。判断道理与之前的碰撞原理类似。举例来说，若是行与行之间的变换，那么我们首先判断是否摄像机处于正确的棋子高度范围，然后看一下是否在宽度范围内（因为此种变换宽度不变），最后判断是否在行与行的范围内，若都在，那么说明摄像机处于棋子行进路线，要被撞开，作标记，进行相应变换。

至此，其实我们已经不知不觉地完成了第三步：对于会导致碰撞的分量，过滤掉，不更新到摄像机位置上。于是我们完成了碰撞检测，当我们在棋盘中漫游的时候，不会飞出场景，可能会撞到棋子，也可能会被行进的棋子撞开，这让我们的游戏体验更加真实了。

Chapter3 光照系统

图 6 两方棋子

在我们的棋盘场景中，载入的模型，都没有加载贴图。我们选择的是直接把红黑双方的棋子指定为带一定透明度的红色和深灰色。这样可以让整个棋盘上双方的棋子容易分辨，呈现出统一的对抗气氛，同时制造一种全息投影的质感。

基于这样的简洁设计理念，我们的光源的主要任务是提供一个照明的功能。所以我们在光照系统上，只做了比较简单的冯氏光照模型，在“楚河“的正上方引入了一个点光源照亮整个棋盘空间，实现一定光照模拟效果。

为了让我们的光照系统能够正常运作，法线，观察点等信息是不可缺少的。

对于.obj 模型文件来说，法线已经在模型中生成好了，直接传入着色器即可。而对于自建模型来说，法线都需要我们准备好。无论是我们的基本体素，棋盘还是 nurbs 曲面建模出来的汽车，亦或是水面网格。我们都正确的生成了每个点的法线，并且最后都送到了片段着色器用于光照计算。

而我们的 Camera 对象也提供了一个接口，接受一个 Shader 对象的指针，将当前的观察点更新到着色器程序中去。

上述细节本章按下不表，在模型模块和摄像机的介绍时你应该都可以找到它们。

代码实现的光照系统核心模块，在 GPU 端是我们的片段着色器代码，在 CPU 端是 Light 类。这两个部分一起完成了光照系统的构建。

我们会在本章介绍片段着色器和 Light 类的设计细节，同时会讨论点光源衰减的实现，最后我们会介绍我们的材质系统。

片段着色器

我们在片段着色器构造了这样的两个结构体，分别设定材质的属性和光照的属性。

材质中的两个 sampler2D 是用来采样贴图的。下面两个 bool 表征是否有相应的贴图。在没有贴图的时候，光照渲染会采用下面设置的三个纯色向量。shininess 是材质的反光度。

光源属性中，我们只简单的封装了点光源和平行光源。bool 变量 pointlight 指示是否是点光源。position 和 direction 分别供点光源和平行光使用。ambient,diffuse,specular 三个分量则指示光源的属性。

```
struct Material
{
    sampler2D texture_diffuse0;
    sampler2D texture_specular0;
    bool texture_diffuse0_exist;
    bool texture_specular0_exist;
    vec4 texture_diffuse_color;
```

```
    vec4 texture_specular_color;
    vec4 texture_ambient_color;
    float shininess;
};
struct Light
{
    vec3 position;
    vec3 direction;
    vec3 ambient;
    vec3 diffuse;
    vec3 specular;
    bool pointlight;
};
```

根据材质和光照，分别计算环境光，漫反射，镜面反射三项反射分量。最后相加即为片段颜色。

```
vec3 ambient，diffuse，specular;
if(material.texture_diffuse0_exist)  ambient=
light.ambient*texture ( material.texture_diffuse0 , gs_out.TexCoords).rgb;
else ambient = light.ambient * material.texture_ambient_color.rgb;
//环境光
vec3 norm = normalize(gs_out.Normal);
vec3 lightDir;
if(light.pointlight) lightDir=
normalize(light.position - gs_out.FragPos);
else lightDir = normalize(light.direction);
float diff = max(dot(norm, lightDir),0);
if(material.texture_diffuse0_exist)diffuse=
light.diffuse * diff * texture(material.texture_diffuse0, gs_out.TexCoords).rgb;
else diffuse = light.diffuse * diff * material.texture_diffuse_color.rgb;
//漫反射
vec3 viewDir = normalize(viewPos - gs_out.FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), material.shininess);
if(material.texture_specular0_exist)specular=
light.specular * spec * texture(material.texture_specular0,gs_out.TexCoords).rgb;
else specular = light.specular * spec * material.texture_specular_color.rgb;
//镜面反射
vec3 result = ambient + diffuse + specular;
FragColor = vec4(result, material.texture_diffuse_color.a);
```

这里要声明的是，这仅仅是模型渲染部分的片段着色器代码。由于不同的对象的需求不同，模型，水面，光源对象，棋盘，天空球分别都有自己的片段着色器。但是它们都大同小异，光照模型的基本框架都是和模型渲染用的片段着色器一样的。

Light 类

在 CPU 程序中，Light 对象起到封装光源的作用。它组织的数据对象和 GPU 中片段着色器的光源数据有着相似的格式。

在 CPU 程序中，实例化一个 Light 对象后，光源信息就全部格式化的存储好了。Light 对象提供了接口，允许用户把光源信息更新到着色器中，只需要调用 setLight 即可。同时 Light 对象会自动的为点光源创建一个光源模型。调用 draw 函数可以把光源本身绘制出来。

```
class Light
{
private:
    bool  pointLight;
    glm::vec3 pos,ambient,diffuse,specular;
    Shader *lightShader;
    Geometry * mylight;
    void buildModel();
public:
    Light(glm::vec3 pos, glm::vec3 ambient, glm::vec3 diffuse, glm::vec3
specular,
          bool pointLight);
    void setLight(Shader *myShader);
    void draw();
};
```

更加真实？—— 光源衰减

虽然我们的场景风格对光源的真实性要求不高,但是我们还是发现原有的光照系统效果不是很好。

日常生活中，点光源是距离越远亮度越低的。一个点光源，应该是离得近的一圈比较亮，远的地方逐渐安暗下来。

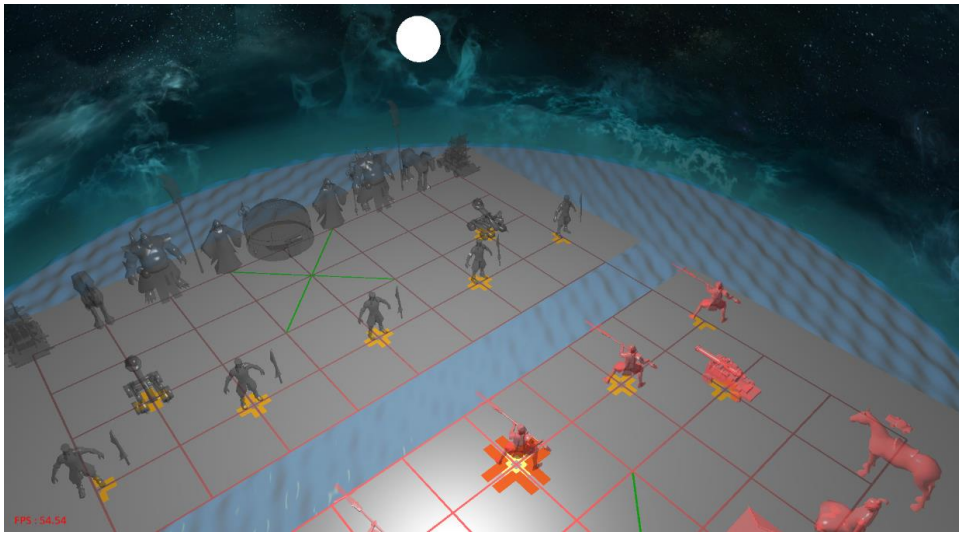


图 7 原本光照效果

但是从上图可以看到，除了反射处有较强的高光，周围的亮度几乎都无差别。看起来有一种违和感。所以我们在片段着色器中引入了光源衰减的计算。

光源衰减的计算，我们采用了一个非线性的表达式：

$$F_{att} = \frac{1.0}{K_c + K_l * d + K_q * d^2}$$

其中 Kc 一般取常数 1，这样可以保证F_{att}总是不超过 1，即光强不会放大。d 是指点和光源的距离。K_l是距离的线性系数，K_q是距离的二次系数。

要完成衰减计算，我们只需要预设好下面三个 K 常数，然后在片段着色器最后的 result 向量那里乘一个F_{att}就可以了。

所以我们在片段着色器中要为光源的结构体加上两个成员：

vec3 attenuation; bool toAttenuation;

表征是否带光源衰减和光源衰减使用的三个系数。

同时我们要在计算光源代码的末尾添加如下的代码：

```
float distance=length(light.position-gs_out.FragPos);
float attenuation=1.0;
if(light.toAttenuation)attenuation=1.0/
(light.attenuation.x+light.attenuation.y*distance+light.attenuation.z*distance*distance)
vec3 result = (ambient + diffuse + specular)*attenuation;
```

同样，CPU 端的 Light 类中也应该增加对这一属性的维护。我们重载了构造器，并且也为其增加了判定是否衰减的 bool 变量以及衰减系数的向量。这样当调用 setLight 时，如果设置了光源衰减，光源衰减系数也会被送到着色器中。

可以看到，添加了光源衰减后，光强开始随距离增加而衰减了，对比度出现了。

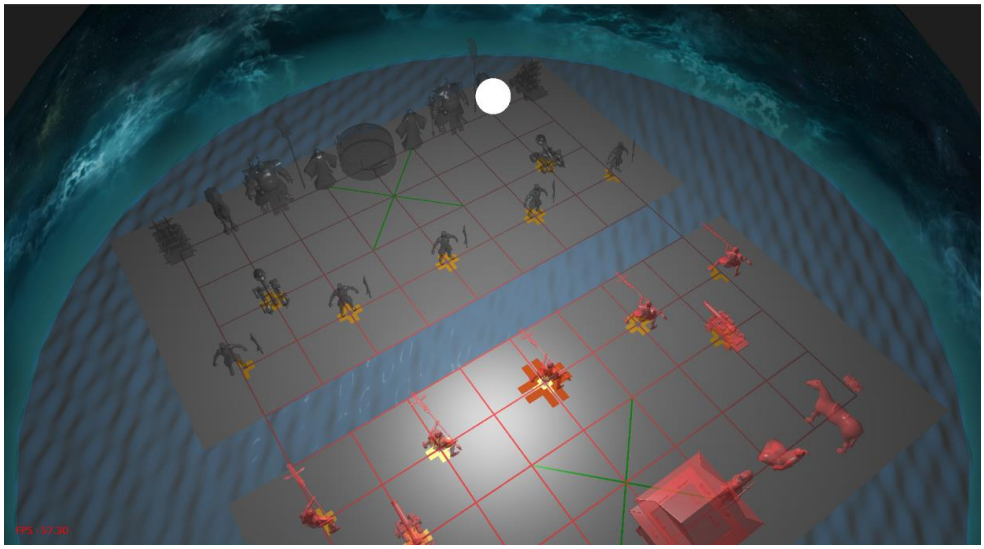


图 8 改良后的光照效果

材质系统

材质部分的封装，具体的代码可以在 TEXTURE.h 中找到。三个类中 Skybox 是以前天空

渲染版本遗留的，我们还是保留它，但是在这个项目中没有用到。我们主要介绍 MyTexture 类和 Material 类，前者是贴图对象的封装，后者是材质对象的封装。

（1）MyTexture

```
class MyTexture
{
private:
    GLuint ID;
public:
    MyTexture(string s,int type=0)
    {
        glGenTextures(1, &ID);
        glBindTexture(GL_TEXTURE_2D, ID);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        int width, height, nrChannels;
        stbi_set_flip_vertically_on_load(true);
        unsigned char *data = stbi_load(s.c_str(), &width, &height, &nrChannels, 0);
        if (data)
        {
            if (type == 0)glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB,
            width, height, 0, GL_RGB, GL_UNSIGNED_BYTE, data);
            else glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, width, height, 0, GL_RGBA,
GL_UNSIGNED_BYTE, data);
            glGenerateMipmap(GL_TEXTURE_2D);
        }
        else
        {
            std::cout << "Failed to load texture" << std::endl;
        }
        stbi_image_free(data);
    }
    GLuint getID()
    {
        return ID;
    }
};
```

纹理对象的构造器接受一个纹理图片路径，使用 stb_image 图形库加载成纹理要求格式的 byte 类型串。调用 OpenGL 相关函数，产生纹理对象。一个 GLuint 类型的变量记录纹理的 ID。绑定纹理时，我们从纹理对象获得 ID 传入着色器即可。

（2）Material

```
class Material
{
private:
    static map<string, Material*>materialTable;
    static bool hasInit;
    glm::vec3 ambient;
```

```
    glm::vec3 diffuse;
    glm::vec3 specular;
    float shininess;
    float alpha;
public:
    static void initMaterial();
    static Material *getMaterialByName(string name);
    Material
    (glm::vec3 ambient,glm::vec3 diffuse,glm::vec3 specular,float shininess,float alpha);
    void setMaterial(Shader *myShader);
};
```

Material 类的静态成员用来处理我们预设的 24 种材质。这些材质参数是一些前人根据经验整理出来的。我们用一个 map 容器来存放材质名称/参数对。提供了静态的 initMaterial 接口从我们的 material.txt 文档种导入这些预设材质。提供了静态的 getMaterialByName 接口允许通过名称获得预设的材质对象。

材质对象的构造器接受自己的若干个参数存储起来。提供一个接口 setMaterial，接受一个着色器对象的指针，将自己的材质参数更新到片段着色器的 Material 结构中去。

Chapter4 模型导入系统

复杂的模型对象很难手工的设置每个顶点来建模，为它们预设好法线，纹理坐标也是相当复杂的事情。我们更加常做的是把模型直接导入到程序中。复杂的模型通常都是由 3D 艺术家在 Blender,3DS Max,Maya 这样的工具中精心制作的。

3D 建模工具可以让艺术家创建复杂的形状，并且可以通过一种叫做 UV 映射的手段来应用贴图。这些工具导出模型的时候，会自动把顶点坐标，法线，纹理坐标生成好。这样可以在不了解图形技术细节的前提下就构造出复杂和精致的模型。

图形开发者要做的是解析这些导出模型文件，提取有用信息，存储为 OpenGL 能够理解的格式。模型的文件格式很多，都有自己导出数据的方式。使用现有的导入库可以帮助我们导入多种格式的模型文件。我们在使用 Assimp 库的基础上，封装了一套自己的模型导入系统，用我们自己的数据结构来组织模型。

1. Assimp

Assimp （Open Asset Import Library）是一个非常流行的模型导入库，它会把模型数据加载至 Assimp 的通用数据结构中，不同格式的模型，加载后，都可以用同样的抽象接口从数据结构中提取我们想要的数据。

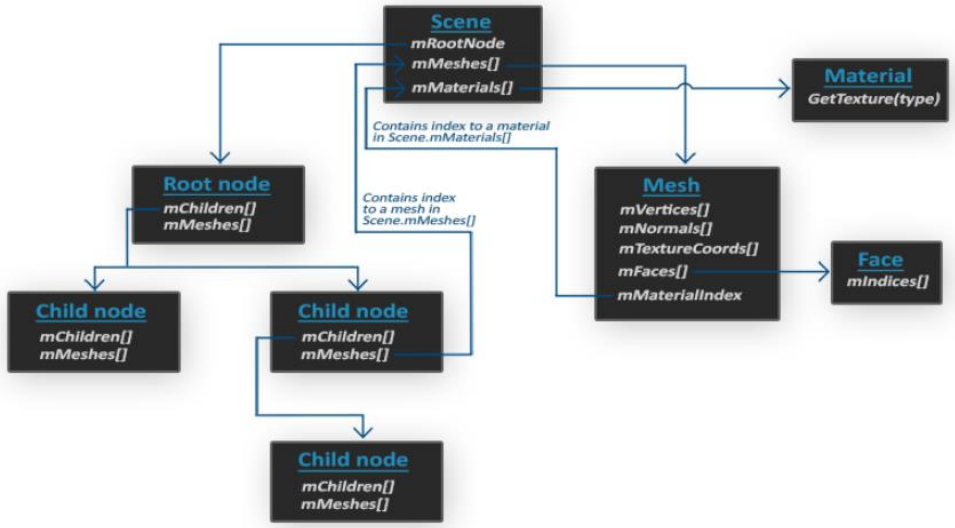


图 9　Assimp 的数据结构，参考自 LearnOpenGL CN

导入一个模型的时候，整个模型会加载进一个场景(Scene)对象中。顶点数据，材质，网格，所有的模型/场景数据都在 Scene 对象当中。

Root node 是存在 Scene 对象中的一颗子树，存的是一系列索引，指向 Scene 中的 mMeshes 数组中的网格数据。真正的 Mesh（网格）对象存在 mMeshes 数组中（mMeshes 也是 Scene 对象的成员）。

Mesh 对象本身包含了渲染自身需要的所有数据，如顶点，法向量，纹理坐标，面，材质。一个网格包含了多个面。(对复杂的物体建模，都是先对各个部分，分别先建模，最后再组装起来。每个单独的最小组成体，就叫做一个网格。网格是模型的一个较底层的抽象，再往下就对应的就直接是顶点了。)

网格中的 Face 代表的是渲染图元，每个渲染图元又包含了组成图元的顶点的索引。网格中也又 Material 对象，这个对象提供了接口可以让我们获得对应材质的各种属性，如颜色和纹理贴图等。

如图，Scene 有三个直接的成员。第一个是一颗索引树，这棵树组织了我们的模型结构，

它里面存了各个成分的对应内容的索引。第二个是我们的网格数组，里面存了网格的实际数据。第三个是材质数组，我们用到的素材都被放在里面。

网格的封装

上面已经说过了，数据的最小抽象体就是网格。网格对应的材质可以从 Material 中去拿，自己渲染所需的内容已经自我封装了。我们沿着模型树去把每个网格都渲染出来，一个模型对象就出来了。所以网格是我们实际上要去着手封装的底层抽象对象。我们需要一个自己的网格类来管理我们的网格对象，并组织它的绘制。

```
struct Vertex
{
    glm::vec3 Position;
    glm::vec3 Normal;
    glm::vec2 TexCoords;
};
struct Texture
{
    unsigned int id;
    string type;
};
class Mesh
{
public:
    vector<Vetex>vertices;
    vector<unsigned int>indices;
    vector<Texture>textures;
    Mesh(vector<Vertex>
    vertices,vector<unsigned int>indices,vector<Texture>textures);
    void Draw(Shader shader);
private:
    unsigned int VAO,VBO,EBO;
    void setupMesh();
}
```

运行构造函数时，就把网格的所有数据赋给网格对象。我们在 setupMesh 中初始化对应的缓冲，如 VBO,VAO,EAO。最后用 Draw 函数来绘制这个网格。我们通过把着色器丢给 Draw 函数来完成定向的着色器绑定。

对象的读入，初始化缓冲都非常平凡无奇。

渲染这一步，涉及到了网格的各个成员，这会让我们遇到几个问题。

如何绑定纹理?

我们一开始并不知道网格有多少纹理，纹理的类型。如何在着色器中设置纹理单元和采样器?

一个比较简单粗暴的解决方案，事先按照命名规范为漫反射纹理和镜面反射纹理分别预定义若干个纹理采样器。这样就算有多个纹理，我们也可以直接用对应的名字引用到采样器中。

```
uniform sampler2D texture_diffuse1;
uniform sampler2D texture_diffuse2;
uniform sampler2D texture_diffuse3;
uniform sampler2D texture_specular1;
uniform sampler2D texture_specular2;
```

PS:这样做很直观，但是可能会有点浪费绑定和 uniform 调用。

```
void Draw(Shader shader)
{
    unsigned int diffuseNr = 1;
    unsigned int specularNr = 1;
    for(unsigned int i = 0; i < textures.size(); i++)
    {
        glActiveTexture(GL_TEXTURE0 + i); // 在绑定之前激活相应的纹理单元
        // 获取纹理序号（diffuse_textureN 中的 N）
        string number;
        string name = textures[i].type;
        if(name == "texture_diffuse")
            number = std::to_string(diffuseNr++);
        else if(name == "texture_specular")
            number = std::to_string(specularNr++);

        shader.setFloat(("material." + name + number).c_str(), i);
        glBindTexture(GL_TEXTURE_2D, textures[i].id);
    }
    glActiveTexture(GL_TEXTURE0);

    // 绘制网格
    glBindVertexArray(VAO);
    glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, 0);
    glBindVertexArray(0);
}
```

这里我们统一绘制的是三角片元。三角片元是组成我们的图形对象的最常见的一类基本图元。

模型的封装

前面我们封装了网格类，网格类是一个比较底层的抽象，一般的模型都是若干网格的集合体。所以我们需要再用一个更加顶层的抽象来封装模型。我们用 Model 类来做了这个封装，一个 Model 的实例对象中，存了若干 Mesh 类的实例对象。层层封装，最后我们实现了像 Model 类构造器传入一个文件路径，就自动化加载模型的引擎。

（1）Model 类

```
class Model
{
public:
    Model(char *path)
    {
```

```
        loadModel(path);
    }
    void Draw(Shader shader);
private:
    vector<Mesh>meshes;   //网格成员
    string directory;
    void loadModel(string path);
    void processNode(aiNode *node,const aiScene *scene);
    Mesh processMesh(aiMesh *mesh,const aiScene *scene);
    vector<Texture>loadMaterialTextures
    (aiMaterial *mat,aiTextureType type,string typeName);
}
```

vector meshes ： 存储作为组成部分的网格对象。
构造器接受一个模型文件路径，通过 loadModel 方法载入到 model 对象中。
Draw 函数比较平凡，就是遍历所有的网格，调用它们的 Draw 函数而已。

（2）loadModel

使用 Assimp 来加载模型，模型会被加载到一个叫做 scene 的数据结构中。scene 是 Assimp 数据接口的根对象。获得这个对象后，我们就能访问到加载后的模型中所有所需的数据。

```
Assimp::Importer importer;
const aiScene *scene
= importer.ReadFile(path,aiProcess_Triangulate | aiprocess_FlipUVs);
```

声明一个 Assimp 命名空间中的 Importer 对象，然后调用它的 ReadFile 方法。这个方法第一个参数是文件路径，第二个参数是一些后期处理的选项。aiProcess_Triangulate 告诉 Assimp,如果模型不是全部由三角形组成的，就对那些非三角图元三角化。aiProcess_FlipUVs 将会在处理的时候翻转 y 轴的纹理坐标，OpenGL 中大部分图像都是反的。

ReadFile 方法把模型加载和处理后，会放到一个 aiScene 对象中，返回这个对象的地址。我们接收这个地址就可以了。

完整的 loadModel

```
void loadModel(string path)
{
    Assimp::Importer import;
    const aiScene *scene =
    import.ReadFile(path,aiProcess_Triangulate | aiProcess_FlipUVs);
    if(!scene|| scene->mFlags & AI_SCENE_FLAGS_INCOMPLETE || !scene->mRootNode)
    {
        cout<<"ERROR::ASSIMP:: "<<import.getErrorString()<<endl;
        return;
    }
    directory = path.substr(0,path.find_last_of('/'));
    processNode(scene->mRootNode,scene);
}
```

加载模型后，立即做异常检查。检测指向场景对象的指针是否为空？检查场景是否完整的加载了？检查场景的根节点是不是为空？如果出错了，做错误报告并返回。
用 path.substr(0,path.find_last_of('/'));来获得文件路径的目录路径。
如果一切正常，我们就会着手处理场景中的所有节点。

processNode 方法会处理这个过程。

（3）processNode

第一个参数接收场景的节点，第二个参数接受场景对象。（节点中的数据就是场景对象中数据对象的索引）

因为对象节点实际上是组织成的一棵树形结构，我们的 processNode 也会被组织成一个递归形式。

```
void processNode(aiNode *node , const aiScene *scene)
{
    for(unsigned int i=0; i<node->mNumMeshes;i++)
    {
        aiMesh *mesh = scene->mMeshes[node->mMeshes[i]];
        meshes.push_back(processMesh(mesh,scene));
    }
    for(unsigned int i=0;i<node->mNumChildren;i++)
    {
        processNode(node->mChildren[i],scene);
    }
}
```

节点处存了若干个网格索引。分别在 scene 中取得对应的网格对象。返回的指针送到 aiMesh *mesh 处。然后通过 processMesh 方法，把 scene 对象中的网格数据加载成我们在 OpenGL 中自己定义的 mesh 对象的格式，然后返回一个 mesh 对象，放到 model 对象的网格容器中。

这里我们发现我们其实也可以直接对 Scene 对象的网格数组直接访问，没必要去回溯？值得一提的是这样一套树形管理结构，是为了给网格之间定义一个父子关系。有了父子关系，如果我们要想让一部分内容做相关的变换，可以直接把命令传递给那一块内容的父网格，然后对整棵子网格树进行同样的操作，就可以实现整体的变换。我们暂时不会用这样的一套整体的系统。但这里仍然用这样一套处理方式。

在处理网格的时候，我们是直接调用的 processNode 方法，我们下面会介绍它。

（4）processMesh

```
Mesh processMesh(aiMesh *mesh , const aiScene *scene)
{
    vector<Vertex>vertices;
    vector<unsigned int>indices;
    vector<Texture>textures;
    for(unsigned int i=0;i<mesh->mNumberVertices;i++)
    {
        ……
        处理相关的顶点数据
    }
    if(mesh->mMaterialIndex>=0)
    {
        ……      处理材质
    }
    return Mesh(vertices,indices,textures);
}
```

关于网格的材质，需要说明一下。
网格只包含了材质的索引，因为材质是单独的存在 scene 中的。

```
if(mesh->mMaterialIndex>=0)
{
    aiMaterial *material = scene->mMaterials[mesh->mMaterialIndex];

    vector<Texture>diffuseMaps                                                    =
    loadMaterialTextures(material,aiTextureType_DIFFUSE,"texture_diffuse");
    textures.insert(textures.end(),diffuseMaps.begin(),diffuseMaps.end());

    vector<Texture>specularMaps =
    loadMaterialTextures(material,aiTextureType_SPECULAR,"texture_specular");
    textures.insert(textures.end(),specularMaps.begin(),specularMaps.end());
}
```

根据索引获得 aiMaterial 对象。然后用 loadMaterialTextures 方法(这个方法也是我们自己实现的)从 aiMaterial 对象中读取纹理。这个方法最后会返回一个装 Texture 结构的容器。然后我们调用 vector 的方法，把返回的容器中的内容添加到 textures 容器中。

（5）loadMaterial

遍历给定纹理类型的所有纹理位置，获取纹理的文件位置，加载和生成纹理，存储在 Vertex 结构体中。(这里回顾一下 texture 的管理：gentexture 方法开辟一块存储纹理信息的空间，返回这个纹理存储块的 ID，然后我们把这个 ID 绑定在纹理对象上，然后对纹理对象进行我们的操作。最后解绑后，相应的纹理信息就写到了纹理 ID 对应的存储块中。我们只需要用 ID 在上下文中引用纹理即可。)

```
vector<Texture>loadMaterialTextures(aiMaterial *mat,aiTextureType type,string
typeName)
{
    vector<Texture>textures;
    for(unsigned int i=0;i<mat->GetTextureCount(type);i++)
    {
        aiString str;
        mat->GetTexture(type,i,&str);
        Texture texture;
        texture.id=TextureFromFile(str.C_str(),directory);
        texture.type=typeName;
        texture.path=str;
        textures.push_back(texture);
    }
    return textures;
}
```

读取存在 scene 对象中的材质对象中对应纹理类型的纹理数量。然后用 type，索引，从这个材质对象中获取纹理信息，放在一个类型为 aiString 的字符串中。这个字符串实际上存的就是这个对应纹理文件的位置。但是它是一个相对于本地目录的位置，我们把之前处理号的模型文件目录路径与文件路径一起传给一个 TextureFromFile 方法，在这个方法中用我们的 stb_image 库来加载对应纹理。

PS：我们假定了模型文件中纹理文件的路径是相对于模型文件的本地路径。有的时候，一些模型中也会出现绝对路径。这需要我们手工的调整。

（6）纹理加载的优化

我们的 Texture 结构：struct Texture{unsigned int id;string type;aiString path;};有一个 path 成员来存路径。

但其实我们是先根据路径加载了纹理直接放到 Texture 结构中的。这样的存储意义何在？这是因为一般的模型中，许多网格都会重用同样的纹理。如果每次加载一个网格就把所有的纹理都加载一遍，会有大量的重复加载过程。加载纹理是一个开销不小的流程，这会带来很大的性能瓶颈。

一个好的解决方法就是，把加载过的路径都存下来，每次要加载新的纹理时，先看一看这个路径是否已经被加载过了，如果已经加载过了就不需要再加载了，直接定位到之前加载的纹理即可。

这个存储可以使用 vector 或者 map 容器。vector 容器每次都需要线性扫描，map 则是用一颗红黑树维护，有更低的访问时间复杂度。

但是这取决于贴图量。根据我们的经验。很多简单的模型，贴图素材的数量都是个位数，map 就显得太“重”了。

Chapter5 棋盘场景设计

外部棋子模型素材的筛选

对于一个场景来说，模型素材的挑选非常重要，合适的模型，会让整个场景更有感染力。与环境和主题格格不入的模型则会严重破坏场景的视觉感官。所以为统一与契合中国象棋的主题，结合历史，我们在双方选择了使一方为汉风格，一方为契丹风格。

在选择的过程中，我们为了使得程序运行正常迅速，每个 obj 模型的大小都控制在了 3M 以下；并且双方的不同种棋子都是不相同的，以体现多样性，提高观赏性。

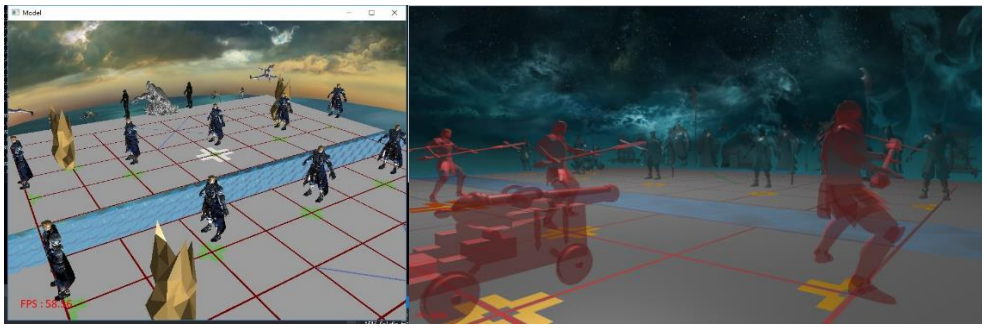


图 10　左图为测试棋盘逻辑时随意放置模型的场景，右为模型统一风格后的场景

环境设计

图 11　天空环境

天空环境的加入不仅可以极大的增强场景的宽阔感，还可以很好的渲染场景的氛围。我们用一个指定了贴图的巨幅半球面来渲染天空环境。这个半球面模型是.obj 格式的，一些 artist 已经为我们完成了贴图的指定，并且配套的贴图素材包也提供给我们了。我们只需要在.mtl 文件里面修改贴图文件路径，就可以随意的更换天空环境贴图。我们最终选择了上面这款贴图，幽绿的星云背景，提供了一个阴暗诡谲的气氛，契合象棋博弈过程中幕后运筹谋划，尔虞我诈的斗争哲学。

地面环境设计上，我们就直接让整个棋盘悬浮在一片波动的水面上，双方相隔的楚河波涛汹涌。一眼望去尽是水，这符合我们简洁风格设计的初衷。用一片水作为地面环境，不仅可以省去一大片花里胡哨的细节，同时也能提供一种与世隔绝的开阔感。水面的材质设计上，我们用了比较卡通的风格，同时赋予了较强的振幅，看起来有较强的波动感。

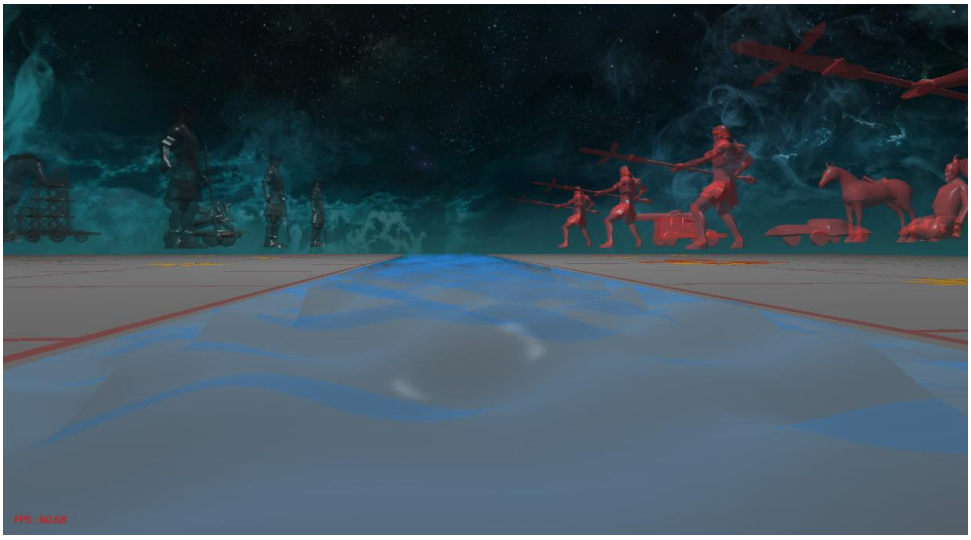


图 12　水面展示

在光照系统一章中，我们已经提过了光源设计的思路。我们简单的用一个悬于楚河正上方的球形点光源来为整个场景提供光照环境。

棋盘设计

棋盘主要可以分为盘身，盘纹两部分。盘身就是两个矩形块，我们为它赋予之前预设的材质包中的银质属性。

盘纹包括纵横交叉的线条和主将所在的四格区域上的×型线以及卒和炮初始放置位的十字型标纹。这些对象底层都是矩形片。线条是长窄的矩形片，十字型标纹则是八块大小相同的矩形片拼凑而成的。

在概述中，我们已经谈了相应的封装结构。底层的矩形片被抽象为 Square 类，它统一管理矩形的建模，材质，光照，位置等属性。×型线和十字型标纹都被抽象成更高级的对象，封装到了 Cross 类和 Tag 类中。

在 chessBoard 对象中，我们统一实例化了银质的盘身，金质的十字标纹，翡翠材质的 x 型线，红色橡胶材质的纵横棋道。最后组成的完整棋盘就被严实的封装在了 chessBoard 对象中，调用一个 draw 函数方法就可以全部绘制出来。。

除了棋盘元素之外，我们还要考虑供玩家操控的光标。我们用一个更大的十字标纹来表示这个光标，并赋予了铜材质，用来和棋盘上的十字盘纹区分。选定棋子，按下回车后，出现玉质的方形块进行高亮标记，选定移动目的后消失。

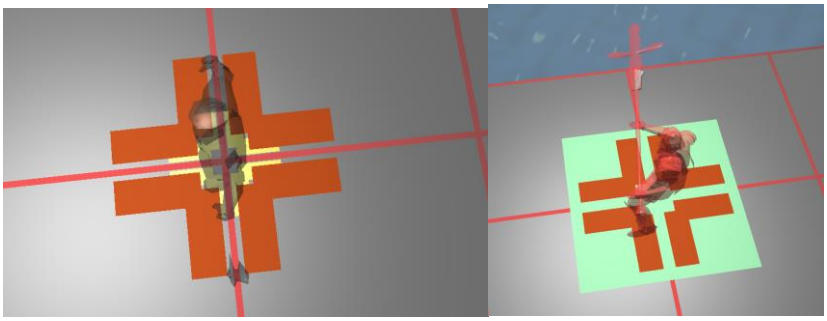


图 13　选取光标

关于用户的控制这一块，我们都完整的封装在了 chessManager 类中，它提供了光标的操控，棋子选定，通知棋子移动，用户键盘检测等功能。

动画效果

棋子的移动，棋子的消灭等，我们都设置了动画效果。采用多帧连续绘制，让棋子的移动和消散变成一个动态的过程，而不是在下一帧立刻消失。

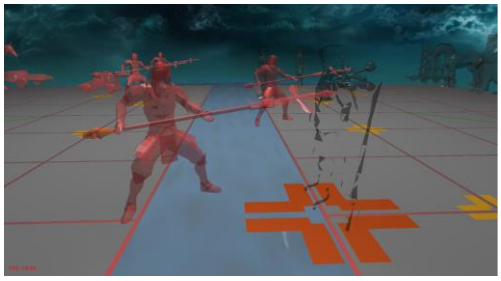


图 14　消散效果

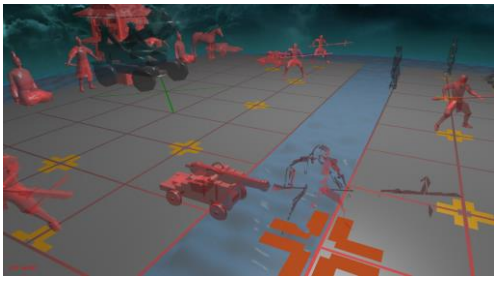


图 15　跳跃与消散效果

我们为棋子的移动提供了一秒钟的渲染，棋子的坐标在这一秒钟内逐帧变化。最后渲染出来的效果就是一个动态的移动过程。考虑到马，象，炮这三个可能会有非直线运动的情况，我们还设置了跳跃动画，高度的变化是一个关于时间的 sin 函数。

同时对于被消灭的棋子，我们单独用一个带几何着色器的着色器程序进行渲染。在几何着色器阶段，模型的顶点会随着法线向外发散。这样就会产生一种模型逐渐向四周碎裂的动画效果。一秒结束后，棋子落到目的点，被消灭的棋子消失，不再被渲染。

如图，左为被消灭的黑方卒正在消散，右为跳跃在半空中的黑炮和消散中的红兵。

Chapter6 棋盘逻辑

棋盘逻辑是我们的象棋游戏得以运行的主要框架。它负责和用户进行交互，同时要管理棋盘上行棋的规则，做胜负判定。还要负责棋子模型的管理和放置，棋子动画效果的多帧连续绘制等。本章我们会介绍几个重要的棋盘逻辑管理类。

ChessManager

```
class chessManager
{
private:
    GLFWwindow* window;
    int frontDeltaRow, frontDeltaCol;
    int rightDeltaRow, rightDeltaCol;
    chessGo *myChess;
    float gap;
    Focus *myFocus;
    waterFlow *myWaterFlow[2];
    void updateDirSystem();
public:
    chessManager(float gap);
    void processInput();
    void display();
}
```

chessManager 是管理棋盘逻辑的一个顶层模块。光标是用户和棋盘状态进行交互的唯一接口。所以 chessManager 做的就是实时的检测用户的鼠标和键盘事件。然后通知自己管理的光标对象去落实，光标对象再做出相应的响应，然后通知自己管理的 chessGo 对象去执行，chessGo 做出响应，再通知自己管理的 chessModel 对象。。。。。整个棋盘逻辑就像一个自上而下的行政系统，各司其职，让棋盘游戏有条不紊的进行。

chessManager 提供了 processInput 函数，每帧都会被调用，循环的响应用户的键盘输入。用户按上下左右键和回车键的时候，它会响应，并和 Focus 对象交互。

这里会自然的出现一个方向的问题。用户可以在场景中自由的移动，看向的方向不断变换。前后左右对于用户来说，是在不断变化的。我们的 processInput 对于用户的前后左右进行的响应，还取决于用户的朝向。

我们用 frontDeltaRow,frontDeltaCol,rightDeltaRow,rightDeltaCol 来表示对于摄像机来说向前和向右对应的棋盘矩阵方向。updateDirSystem 每帧都会被调用，实时的获取相机的朝向，选取和相机朝向夹角最小的轴作为“前”对应的方向。由于棋盘法线恒定，就可以计算出右方向对应的轴。

Focus

```
class Focus
{
private:
    glm::vec3 trans;
```

```
    int row, col;
    float gap;
    Tag *myFocus;
    Square *chosen;
    chessGo *myChess;
    bool focusComfirmed;
    int locX, locY;
    int targetId;
    int turn;
    Material * focusMaterial;
public:
    Focus(int row, int col, float gap,chessGo* myChess);
    void drawFocus();
    void moveFocus(int dRow, int dCol);
    bool targetConfirm();
    bool focusConfirm();
};
```

Focus 类封装了用户操控的光标。

row，col 对应的是在棋盘矩阵上的坐标。

gap 记录的是相邻格之间的距离。

trans 记录的是平移量。

一个 chessGo 对象的指针用来和管理棋子运动逻辑的模块进行通信。

一个 Square 对象指针 chosen 用来指向我们选中时显示的高亮玉质方块。

myFocus 指向的是表示光标的十字标记，这个标记我们之前封装在了 Tag 类中。

当选中一个棋子的时候，focusComfirmed 会被置为 true。这个时候我们光标选择的下一个位置则会成为棋子的目的地。locX 和 locY 指向了这个目的地。

我们提供的 drawFocus 会完成光标以及高亮块的绘制。moveFoucs 则是其他模块移动光标的接口。

focusConfirm 处理选中棋子的事件，而 targetConfirm 处理选中目的地的事件。这些事件包括一些状态量的设置，但主要是涉及到棋子。

这两个方法会与之前构造该对象时传入的 chessGo 对象进行通讯，获得光标位置处的棋盘状态信息，同时向 chessGo 对象发送移动棋子的命令，chessGo 再解析命令具体的进行底层执行。

chessGo

```
class chessGo
{
private:
    int board[9][10];
    Shader * explode;
    glm::vec2 pos[9][10];
    float gap;
    Shader * myShader;
    Model *modelResource[14];
    chessModel *myChess[33];
    nurbsCar *myCar;
```

```
    bool going;
    glm::vec3 goDir;
    int goId;
    float goTime;
    float sizeoftype[15][6] = { ……};
    int type[33];
    int locX[33];
    int locY[33];
    static glm::vec3 lightDir;
    int explodedTarget;
    bool jump;
    void initialPos();
    void bindModel();
    void initialExplode();
    bool checkRule(int id, int row, int col);
public:
    bool focusLock;
    chessGo(float gap);
    void initialBoard()；
    void go()；
    bool moveChess(int id, int row, int col);
    int target(int row, int col);
    int getType(int id);
    void drawChess();
};
```

chessGo 是整个棋盘逻辑功能最核心的模块。

棋盘上 32 个棋子都是从 1~32 编号的。int board[9][10]记录的是棋盘上每个位置当前的棋子。如果为 0 表示没有,如果为 i,表示这个位置上现在有一个编号为 i 的棋子。

glm::vec2 pos[9][10]记录的是棋盘上每个位置在对应的世界坐标。(没有 y 分量，因为棋盘的高度是确定的)

Model *modelResource[14];总计有 14 总棋子模型,在渲染开始前会预加载到 Model 对象中。

chessModel *myChess[33];

对应着 32 个具体的棋子;棋子的放缩,旋转,移动等调整,全部交给它管理。

nurbsCar *myCar;红方的车是用 nurbs 曲面建模出来的,所以用单独的一个对象来管。

int type[33];表示编号为 i 的棋子的类型。在实现中，我们把被消灭的棋子的 type 变成自己的相反数（负数）来作为它被消灭的标志。

int locX[33] 与 int locY[33]:记录每个棋子的坐标。

当一个 chessGo 对象被实例化后，会自动调用初始化函数，载入模型，初始化棋盘状态，将模型都调整到对应的位置进行渲染。

chessGo 本身不涉及任何对棋子模型进行操作的底层实现。所有棋子模型都由一个 chessModel 对象数组管理。chessGo 仅仅是通过 chessModel 提供的接口，对每一个棋子进行调整。

chessGo 的 drawChess 函数，每一帧都会被调用，它会调用 chessModel 提供的接口，将每一个还存在的棋子都渲染出来。

在 draw 之前，它都会调用 go 函数。我们有一个 bool 类型的名为 going 的变量。当它为 true 时，go 函数才会执行自己的功能。go 函数是我们实现动画，即连续多帧绘制的模块。go 函数中，指定的模型对象的坐标，会在指定的轨迹上逐帧的变换。

chessGo 的 moveChess 接口是触发动画的源头。这个接口接受三个变量 id，row，col，即要将编号为 id 的棋子移动到 row 行 col 列。moveChess 函数会检测这个移动请求的合法性。合法性是通过 checkRule 函数进行检测的。checkRule 封装了象棋游戏的规则，只会对符合规则的移动返回 true。当合法性检测通过后，moveChess 就会把 going 置为 true，并且初始化 goingTime(动画播放时间)，同时根据移动方向，设置 goDir，这指定了水平方向的平移总量。同时，这个函数还会判定棋子是否会跳跃。如果会跳跃，jump 会被置为 true。这样在下一帧绘制时，go 函数被调用时，就会开始逐帧的朝着目标方向开始移动。

go 函数在开始的时候就会检测 going，如果 going 为 false，就会直接返回。当 moveChess 被合法的调用后，going 被置为 true，go 函数就会开始执行它的播放功能。朝着既定的水平方向，每一帧按照 deltaTime 和 goTime 的比例朝着目标方向移动一小段。同时一个 totTime 会记录播放时间。时间到的时候，go 会被置为 false。至于跳跃的效果，我们简单的截取了 sin 函数从 0 到二分之 pai 的这段波峰，y 会随着时间从 sin 函数上取样，这样就形成了一个跳跃的效果。

同时，chessGo 对象在每次棋子模型变化的时候，都会和 Camera 对象交互。将当前棋盘上 32 个棋子的位置信息实时更新到 Camera 中。Camera 获得这些信息后，实时的进行碰撞检测。

chessModel

chessModel 是直接管理棋子模型的类。

```
class chessModel
{
private:
    float rotAngle;
    float scaleFactor;
    Model *chess;
    nurbsCar *nurbs;
    glm::mat4 model;
    bool Red;
    bool nurbsModel;
public:
    chessModel(Model *chess, float scaleFactor, float rotAngle,bool Red,glm::vec3
    *initshift);
    chessModel(nurbsCar *chess, float scaleFactor, float rotAngle, bool Red, glm::vec3
    *initshift);
    void translate(glm::vec3 *trans);
    void draw(Shader *myShader);
};
```

由于我们的棋子模型使用了外部模型文件和 nurbs 曲面细分建模两种。所以我们重载了构造函数，使得它可以多态的接受两种模型，进行管理。

虽然总共有 32 个棋子，但我们为 14 种棋子模型都只加载了一次，放在堆内存中。chessModel 对象获得的只是自己对应棋子类型的模型的指针。每次渲染的时候，我们仅仅是使用了对应模型的顶点信息，旋转角，平移量等都是我们棋子对象自己的，所以还是可以渲染出 32 个棋子。即 14 个棋子模型的数据被 32 个棋子共享。

决定一个棋子模型形态的仅仅只有它模型数据本身和自己的模型矩阵。所以我们封装棋子，其实就是封装的棋子模型和它的模型矩阵。

chessModel 的构造函数接受一个偏移量，允许我们初始化棋子的时候，将他位置进行

微调，调整到正确的朝向和位置上。同时它还提供了 translate 接口，允许我们对棋子的位置随时进行调整。

draw 函数会自动把棋子的信息更新到着色器程序中。外部模块直接调用这个接口就可以渲染它。

Chapter7 基本体素

立方体

每个面两个三角形进行分割即可。

球

根据球的极坐标公式，求坐标，同样重新排序赋值使顶点有右手螺旋三角形的顺序。

圆台

根据圆的极坐标公式，求得一个面的坐标，对 z 方向赋值为 0 和宽度，每两个点之后插入圆中心坐标是之成为右手螺旋顺序画成三角平面片，侧面则分别取两个圆平面的点按顺序排列。

正棱台生成

位于 Geometry 类内部，名为 tpyramid。

需要输入：底面多边形边数 n，上底面外接圆半径 r1，下底面外接圆半径 r2，高 h，坐标（a，b，c），颜色，共 8 个参数。生成棱柱为水平放置，两个底面垂直于棋盘表面。

正棱台基础下的生成

（1）棱柱生成：位于 Geometry 类内部，名为 prisms。

需要输入：底面多边形边数 n，底面外接圆半径 r2，高 h，坐标（a，b，c），颜色，共 7 个参数。

（2）正棱锥生成：位于 Geometry 类内部，名为 pyramid。

需要输入：底面多边形边数 n，底面外接圆半径 r2，高 h，坐标（a，b，c），颜色，共 7 个参数。

（3）正圆锥生成：位于 Geometry 类内部，名为 taper。

需要输入：底面圆半径 r2，高 h，坐标（a，b，c），颜色，共 6 个参数。

Chapter8 Nurbs 曲面建模

B-样条

本项目采用双三次均匀 B-样条进行曲面建模。
根据 B-样条递推公式可以确定三次 B-样条。

$$B_i^{n+1}(x)=\frac{x-t_{i-1}}{t_{i+n}-t_{i-1}}B_i^n(x)+\frac{t_{i+n+1}-x}{t_{i+n+1}-t_i}B_{i+1}^n(x)B_i^0=\begin{cases}1, if\ \ x\in(t_{i-1},t_i)\\ 0, otherwise\end{cases}$$

再根据求 B-样条的曲面公式:

$$P(u,v)=\sum_{i=0,j=0}^{i=k,j=q}p_{ij}B_i^3(u)B_j^3(v),\ k,q\text{为控制点在}u,v\text{方向的数目}\ \ p_{ij}\text{为控制点坐标}$$

可以得到任意 u,v 的计算结果，对 u,v 取网格划分，在本项目中在 u,v 方向划分网格数相同均为 n 个。则得到 n*n 的矩阵，其中存储着 n*n 个点坐标，这 n*n 个点即为我们需要用来画图的顶点。对上述 n*n 个顶点按右手螺旋组成三角形的规则重新赋值到新的数组 car，这个数组则可以直接用来画图。

算法的时间复杂度分析

假设要得到 n*n 个顶点，那么它的规模应为:

$$\sum_{i,j}^{n,n}(\lfloor log_2x_i+log_2y_{i+}log_2z_i\rfloor+2)=2n^2+\sum_{i,j}^{n,n}\lfloor log_2x_i+log_2y_{i+}log_2z_i\rfloor$$

简单的可以表示成:

$$n+\lfloor 3*\max(x_i,y_i,z_i)\rfloor$$

在算法实现过程中，每个点的计算经过了三次 B-样条的迭代计算，以及所有控制点的加权求和，因此，每个点经过了 2 次 B-样条公式的判断，乘法运算共 5 次，加权计算 k*q 次，总共 k*q*5*5 次，再加上赋值操作共计 k*q*5*5+1 次，而之后对其进行顺序重编的赋值和顶点法向量计算，每个点各需要一次运算。

因此 n*n 个点所需要的运算次数之和，即算法的时间复杂度为:

(k*q*5*5+1+2)*n*n 次，

$$O\left(\text{poly}(n+\lfloor 3*\max(x_i,y_i,z_i)\rfloor)\right)$$

是一个多项式时间算法。

Chapter9 水面渲染

背景介绍

在过去的几年中，在 GPU 上运行的水模拟模型在不断向上发展。从正弦函数到比较复杂的函数，使用周期波函数的总和来创建动态平铺凸块图的技术已经相对完善，在这样的基础上，我们可以合理并且有一定真实性地捕获水面的动率与细节。本次实验就在楚河汉界的交界处，使用了 Gerstner 波函数上构造出的水波模型对水进行了建模。

这个系统虽然不是严格的物理模拟，但它确实提供了令人信服的、灵活的、动态的水渲染。

（1）一方面因为模拟完全在 GPU 上运行，所以各部分不需要为 CPU 的使用而斗争。

（2）另一方面因为系统参数确实具有数学物理基础，所以更真实，更不易出错。

原理介绍与公式推导

从主观感性意义上，水波在我们眼中就是光滑，并且在随时间作起伏的二维曲面，此模型就是用波函数叠加，塑造出有规律起伏且随时间变化的曲面函数。其中叠加前的“基函数”就是波函数。而最简单的波函数就是三角函数。我们先从最简单的 sin 函数开始考虑。

给出一个单波的函数：

$$W_i(x,y,t) = A_i sin(D_i(x,y) \times \omega_i + t \times \varphi_i)$$

其中 Ai 是振幅，代表第 i 个波最高点距离中线的距离；
Di 是波在二维平面上传播的方向，此处看做一个常数；
wi 为频率，表征波在二维欧式平面上起伏分布的疏密；
φ_i表示波传递的速度，随着时间的变化，波随着这个参数的快慢进行起伏。
叠加得到式子：

$$H(x,y,t) = \Sigma_i(A_i sin(D_i(x,y) \times \omega_i + t \times \varphi_i))$$

由此函数得出的 P 点位置：

$$P(x,y,t) = (x,y,H(x,y,t)) = (x,y,\Sigma_i(A_i sin(D_i(x,y) \times \omega_i + t \times \varphi_i)))$$

实际上，这样叠加会出现一个问题，就是波峰太圆且波峰与波谷的曲率一致，这在真实世界里是不成立的。

所以为了使得水波更加有真实感，我们需要在原本的基础上作出修正，一个方法是在三角函数的基础上加一个常数然后提高次数。

此处我们用另一种方法，在 P 点的 x 与 y 处加一个可以表征平滑度的项，也就是 Gerstner 波函数：

$$P(x,y,t) = \begin{pmatrix} x + \Sigma_i(Q_i A_i \times D_{i.x} \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i)) \\ y + \Sigma_i(Q_i A_i \times D_{i.y} \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i)) \\ H(x,y,t) \end{pmatrix}$$

$$= \begin{pmatrix} x + \Sigma_i(Q_i A_i \times D_{i.x} \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i)) \\ y + \Sigma_i(Q_i A_i \times D_{i.y} \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i)) \\ \Sigma_i(A_i sin(D_i(x,y) \times \omega_i + t \times \varphi_i)) \end{pmatrix}$$

接下来我们分析 P 点上的法向。

由微分几何知识，显式曲面对 x 求偏导所得向量与对 y 求偏导所得向量张成切平面，其外积即为法向量。

所以有如下过程：

$$
\begin{aligned}
B &= \frac{\partial P}{\partial x} \\
&= \begin{pmatrix} \frac{\partial}{\partial x}(x + \Sigma_i(Q_i A_i \times D_{i.x} \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i))) \\ \frac{\partial}{\partial x}(y + \Sigma_i(Q_i A_i \times D_{i.y} \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i))) \\ \frac{\partial}{\partial x}(\Sigma_i(A_i \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i)) \end{pmatrix} \\
&= \begin{pmatrix} 1 - \Sigma_i(Q_i D_{i.x} D_{i.x} \omega_i A_i \times cos(D_i(x,y) \times \omega_i + t \times \varphi_i)) \\ -\Sigma_i(Q_i D_{i.y} D_{i.x} \omega_i A_i \times cos(D_i(x,y) \times \omega_i + t \times \varphi_i)) \\ \Sigma_i(D_{i.x} \omega_i A_i \times cos(D_i(x,y) \times \omega_i + t \times \varphi_i)) \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
T &= \frac{\partial P}{\partial y} \\
&= \begin{pmatrix} \frac{\partial}{\partial y}(x + \Sigma_i(Q_i A_i \times D_{i.x} \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i))) \\ \frac{\partial}{\partial y}(y + \Sigma_i(Q_i A_i \times D_{i.y} \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i))) \\ \frac{\partial}{\partial y}(\Sigma_i(A_i \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i)) \end{pmatrix} \\
&= \begin{pmatrix} \frac{\partial}{\partial y}(x + \Sigma_i(Q_i A_i \times D_{i.x} \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i))) \\ \frac{\partial}{\partial y}(y + \Sigma_i(Q_i A_i \times D_{i.y} \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i))) \\ \frac{\partial}{\partial y}(\Sigma_i(A_i \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i)) \end{pmatrix} \\
&= \begin{pmatrix} 1 - \Sigma_i(Q_i D_{i.x} D_{i.x} \omega_i A_i \times cos(D_i(x,y) \times \omega_i + t \times \varphi_i)) \\ -\Sigma_i(Q_i D_{i.y} D_{i.x} \omega_i A_i \times cos(D_i(x,y) \times \omega_i + t \times \varphi_i)) \\ \Sigma_i(D_{i.x} \omega_i A_i \times cos(D_i(x,y) \times \omega_i + t \times \varphi_i)) \end{pmatrix}
\end{aligned}
$$

$$
\begin{aligned}
N &= B \times T \\
&= (\begin{bmatrix} B.y & B.z \\ T.y & T.z \end{bmatrix}, \begin{bmatrix} B.z & B.x \\ T.z & T.x \end{bmatrix}, \begin{bmatrix} B.x & B.y \\ T.x & T.y \end{bmatrix}) \\
&= \begin{pmatrix} -\Sigma_i(D_{i.x} \omega_i A_i \times cos(D_i(x,y) \times \omega_i + t \times \varphi_i)) \\ -\Sigma_i(D_{i.y} \omega_i A_i \times cos(D_i(x,y) \times \omega_i + t \times \varphi_i)) \\ 1 - \Sigma_i(Q_i \omega_i A_i \times sin(D_i(x,y) \times \omega_i + t \times \varphi_i)) \end{pmatrix}
\end{aligned}
$$

由法向量 z 值，我们可以设：

$$Q_i = \frac{Q}{\omega_i A_i \times numWaves} (Q \in [0,1])$$

Q 越大，波峰越锋利。此处 Q 的上限是为了使 P 的法向 z 分量不小于 0，也即波峰不会出现回环的情况。

在做程序的时候，我们需要的仅仅是 P 点的坐标与法向量。而这些都可以用上面的式子直接算出。需要注意的是，此处由于 P 点的横向移动，只用 x，y 计算出来的 z 值已经不再严格是曲面高度，但是仍旧是易于计算易于微分的。

最后，在操作中我们还需要设定振幅，方向，频率与波速，使得出现的效果足够真实。

常用的方法是设定波长 L 的均值（与天气有关）与波长 L 和振幅 A 的比值，生成介于该长度一半到两倍之间的随机波长，

再由公式：

$$\omega = \sqrt{g \times \frac{2\pi}{L}}.$$

计算出频率，然后用一个常向量代指方向 D。

在我们的实现过程中，我们直接设定了固定的参数，这也是我们应该提高的地方。

Chapter10 Some other tricks

截屏

我们知道，最后屏幕显示的内容，其实都是来自于显存中的颜色缓冲区。我们在渲染循环中每一帧都会交换颜色缓冲，来实现刷新。所以，我们的截屏，实际上就是从颜色缓冲中取回像素矩阵。OpenGL 提供了这样的接口，允许我们从 GPU 将数据传回 CPU。

glReadPixels(0, 0, scrWidth, scrHeight, GL_BGR, GL_UNSIGNED_BYTE, pPixelData);

调用上述函数后，颜色缓冲中的数据就传到了 RAM 中 pPixelData 指向的 buffer 里了。其中 scrWidth,scrHeight 是当前窗口的像素宽和像素高。

注意到我们使用的是 GL_BGR 而不是 GL_RGB，这是因为传回来的数据是反的。。我们需要反着对应进来。

获得像素的 RGB 阵列后，我们就可以导出最简单的 bitmap 格式的图片了。我们直接从一个三通道的 bmp 图片中截取了开头的一段形式化文件头，存储为 dummy.bmp。这样我们在导出图片的时候，就可以直接先从这个 dummy 文件头里读出来直接写入。我们自己需要写的文件头就只剩下长和宽了。

写入文件头后，我们把 pPixelData 缓冲区里面的内容写到 bmp 文件中即可。导出的 bmp 文件名，我们直接用 time 函数返回的时间戳，这是一种比较有效的防止文件名冲突的方法。

我们把截屏函数实现在了 Toolkit 类中。Toolkit 中有一个 checkScrShotRequest 函数在渲染循环中被循环调用，用来监听键盘事件。当按下 G 的时候，就会截屏，截图文件会存在 screenShot 目录下。

```
static void screenShot()
{
    static const int BMP_Header_Length = 54;
    FILE*    pDummyFile;
    FILE*    pWritingFile;
    GLubyte* pPixelData;
    GLubyte  BMP_Header[BMP_Header_Length];
    GLint    i, j;
    GLint    PixelDataLength;

    // 计算像素数据的实际长度
    i = scrWidth * 3;   // 得到每一行的像素数据长度
    while (i % 4 != 0)      // 补充数据，直到 i 是的倍数
           ++i;                    // 本来还有更快的算法，
                                   // 但这里仅追求直观，对速度没有太高要求
    PixelDataLength = i * scrHeight;

    // 分配内存和打开文件
    pPixelData = (GLubyte*)malloc(PixelDataLength);
    if (pPixelData == 0)exit(0);
    char target[100] = "./screenShot/";
    sprintf_s(target+strlen(target),50,"%d.bmp",time(NULL));
    fopen_s(&pDummyFile,   "./screenShot/dummy.bmp" , "rb");
    if (pDummyFile == 0) exit(0);
    fopen_s(&pWritingFile, target , "wb");
```

```
    if (pWritingFile == 0) exit(0);

    // 读取像素
    glPixelStorei(GL_UNPACK_ALIGNMENT, 4);
    glReadPixels(0, 0, scrWidth, scrHeight,
    GL_BGR, GL_UNSIGNED_BYTE, pPixelData);

    // 把 dummy.bmp 的文件头复制为新文件的文件头
    fread(BMP_Header, sizeof(BMP_Header), 1, pDummyFile);
    fwrite(BMP_Header, sizeof(BMP_Header), 1, pWritingFile);
    fseek(pWritingFile, 0x0012, SEEK_SET);
    i = scrWidth;
    j = scrHeight;

    fwrite(&i, sizeof(i), 1, pWritingFile);
    fwrite(&j, sizeof(j), 1, pWritingFile);
   // 写入像素数据
    fseek(pWritingFile, 0, SEEK_END);
    fwrite(pPixelData, PixelDataLength, 1, pWritingFile);
    // 释放内存和关闭文件
    fclose(pDummyFile);
    fclose(pWritingFile);
    free(pPixelData);
    cout << "successfully catch the new frame!" << endl;
}
```

面剔除

当一个面背对观察者的时候，我们直接将它剔除掉，不进入片段着色器中进行深度测试。这是图形渲染中非常常见的一个技巧。如果熟悉 CS 或者前几年盛极一时的《穿越火线》的话，我们都知道，当我们以观察者模式进入一个墙面后，看回去，墙不见了。这是因为此时这个面的法线和我们的观察方向一致，这个面被判定为背对我们，被剔除了。

我们在初始化 OpenGL 上下问的时候，调用 glEnable(GL_CULL_FACE)即可启用这一功能。这个功能判定图元的方向是基于图元点的排列顺序的，它会认为图元的外法线和点顺序满足右手螺旋规则。

所以，开启这一功能后，我们在建模的过程中，也必须强迫自己注意把点传入 VBO 的顺序要满足右手螺旋规则。启用后，场景中通常会有约一半的片段被过滤掉，有着非常好的性价比。

Chapter11 附录

参考文献

[1]材质参数设置：http://devernay.free.fr/cours/opengl/materials.html
[2]Assimp 数据结构图片：https://learnopengl-cn.github.io/img/03/01/assimp_structure.png
[3]水波模型原理：https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch01.html

代码列表

MYTOOLKIT.h	命名空间封装
SHADER.h	着色器的创建，链接和编译
CAMERA.h	相机类
MODEL.h/MESH.h	标准模型文件加载器
TEXT.h	管理文本对象的渲染，显示 FPS
LIGHT.h	管理光源
SKY.h	天空渲染
TEXTURE.h	管理贴图对象
NURBSCAR.h	nurbs 曲面建模的车模型
WATERFLOW.h	水面模型
GEOMETRY.h	基本几何体的建模
SQUARE.h	棋盘盘面
CHESSBOARD.h	棋盘的顶层对象
CHESSMANAGER.h	管理盘面上可移动物的渲染和事件
CHESSGO.h	管理 32 个棋子与逻辑
CHESSMODELMANAGER.h	存储了每个棋子的 Model 对象
chineseChess.cpp	主程序

objectFragmentShader.sd	渲染模型文件
objectVertexShader.sd	渲染模型文件
objectGeometryShader.sd	渲染模型文件
chessBoardVertexShader.sd	渲染棋盘
chessBoardFragmentShader.sd	渲染棋盘
lightVertexShader.sd	渲染光源
lightFragmentShader.sd	渲染光源
skyVertexShader.sd	渲染天空球
skyFragmentShader.sd	渲染天空球
textFragmentShader.sd	渲染文本
textVertexShader.sd	渲染文本
waterFragmentShader.sd	渲染水面
waterVertexShader.sd	渲染水面

小组分工

漆翔宇：框架结构，棋盘逻辑，光照系统，动画系统，场景搭建等编码与文档说明

张博康：相机系统，碰撞检测编码与文档说明

张睿：NURBS 曲面建模，基本体素中立方体，圆柱，圆台，球的编码与文档说明

代汶利：模型筛选，水波推导文档，基本体素中棱柱，棱锥，圆锥，棱台的编码与文档说明，报告整合