“魁地奇桌球”最终版设计报告

沈斯杰　5130379036

2016. 1. 12

一、 项目简介

(一) 最终版要求

考试大作业在第一次和第二次平时作业的基础上添加更多几何模型，实现粒子动画和渲染效果。具体要求如下：

（1）建模：设计有起伏变化的地形和场景包围盒，并将该场景的实现添加到魁地奇的物理系统中（25 分）；

（2）粒子动画特效：在小球的运动或碰撞过程加入不少于一种的粒子动画特效（例如火星飞溅或光环闪耀）（25 分）；

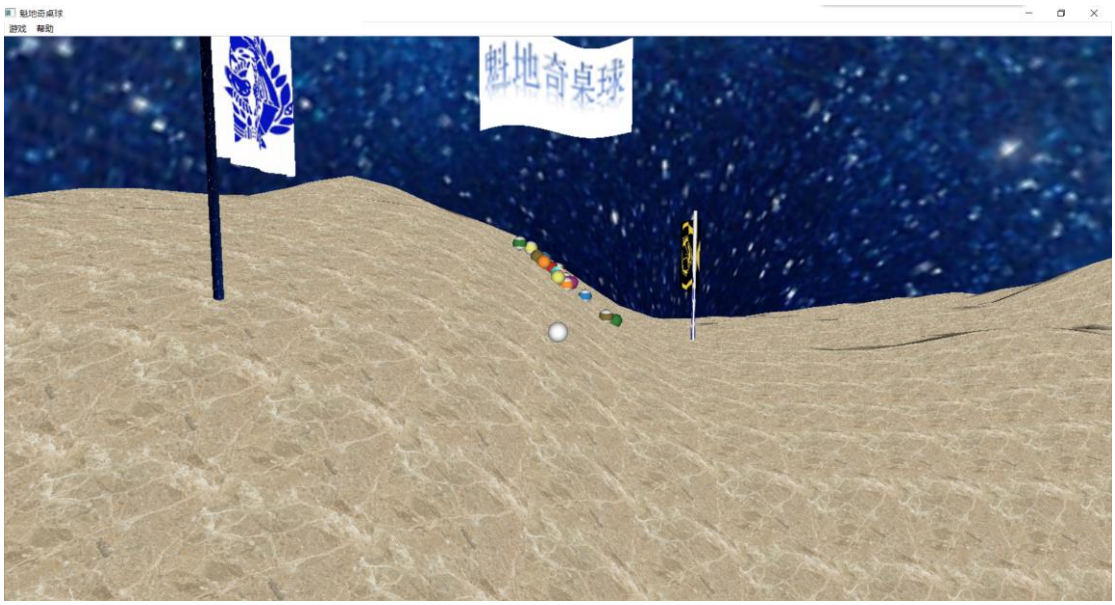
（3）光照：添加场景光照，并添加聚光灯用于照射白色母球并且追随母球（可交互控制灯光的开关）（15 分）；

（4）纹理：使用 perlin 噪声函数添加自然纹理到小球表面上；（25 分）

（5）详细的设计报告以及标准格式的提交文件；（10 分）

（6）实现以上任务以外的建模或特效技术将获得酌情加分。

(二) 游戏截图



(三) 游戏基础设置

1. 游戏的按键设置如下：

按键	功能
W/A/S/D	控制母球移动
J/L	左右移动摄像头
U/O	上下移动摄像头
I/M	前后移动摄像头
K	复原最初位置和视角
G	切换至聚光灯模式

2. 球台上有 15 个球，其中球号为 5/6/7/13/14/15 为移动的“游走球”，彩色球为“飞贼”。初始状态下，“鬼球”会在桌面平面内随机移动，“飞贼”会在桌球及其上空随机移动，经过一段时间后停在桌面上休息，如此循环。
3. 双方阵营有旗帜，以及球台上的标题旗帜。

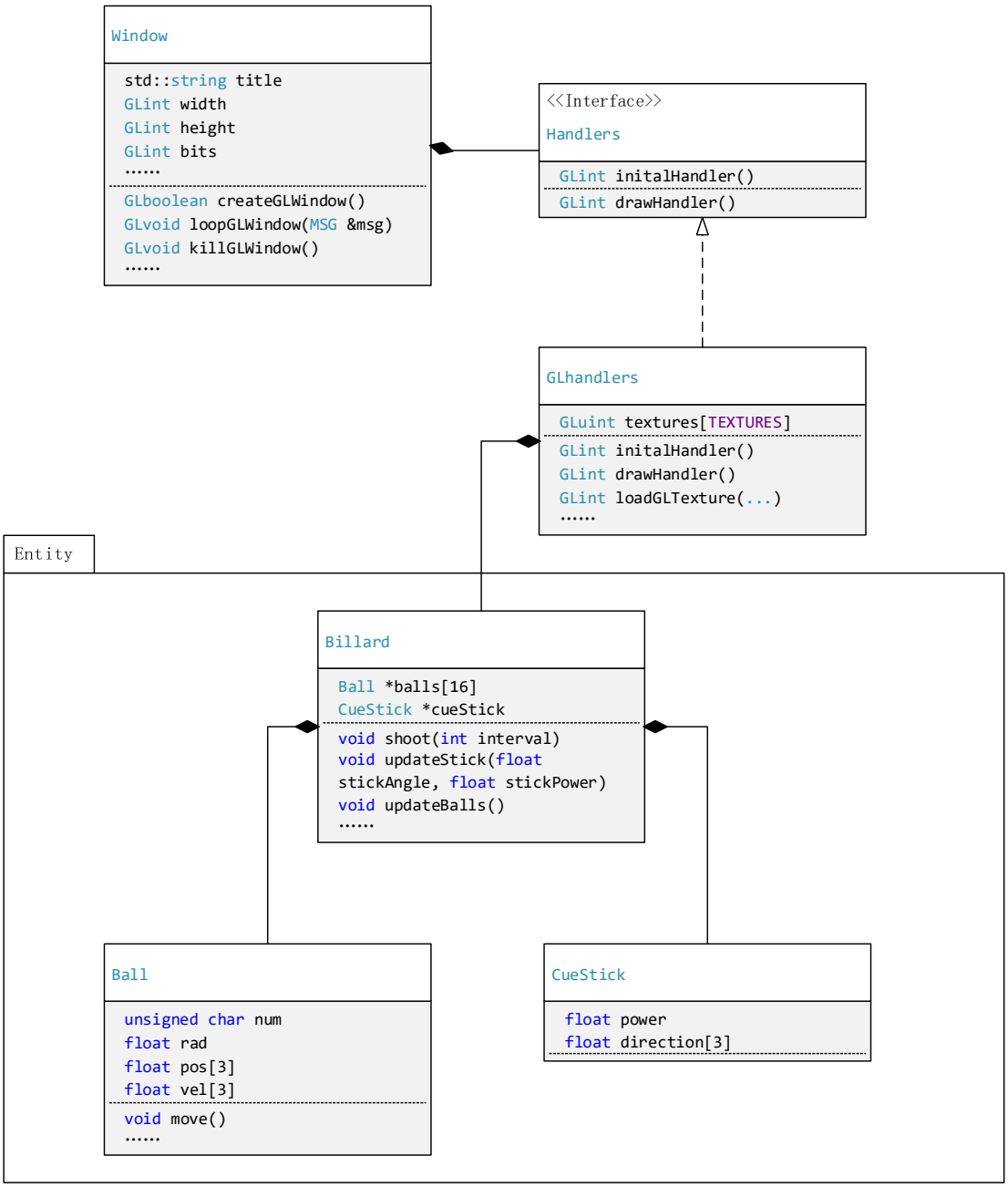
二、 代码结构

在这一版的程序中，主要的架构如下：

头文件	.cpp 文件	职能
background.h	background.cpp	绘制（底层）背景
draw.h	draw.cpp	绘制模型(包括光照等)
	texture.cpp	绘制纹理
entity.h	entity.cpp	球和球杆的实体类
	billard.cpp	游戏逻辑
	main.cpp	初始化和入口函数

进行调整的有：

1. 将原本 main.cpp 中的 initGL()函数移入 draw.h 中作为接口，因为这一部分在逻辑上适合 openGL 的绘制联系在一起的。
2. 本来 billard.cpp 中，对于游戏的逻辑耦合度太高，将球的移动直接在这个文件中实现，现在在实现游戏逻辑中，按照摩擦、力度中改变球的速度，然后调用球这个类中的 move()方法，这样更符合实际情况，耦合度也会降低。
3. 将程序改成面向对象的设计结构，使逻辑更加清晰。大致包图如下（还有一些小类没有画出）。
4. 对球加入部分碰撞效果。



三、 实现效果

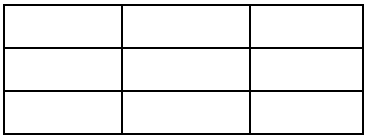
(一) 旗帜的设计

1. 双方阵地的旗帜选用贝塞尔曲面进行设计。关键代码如下：

```
/* For Animation */
for (int i = 0; i < UNUM; i++) {
    for (int j = 0; j < VNUM; j++) {
        ctrlpoints[i][j][0] =
            sin(2 * PI*(-2.5 * i / 14.0f + t / float(T))) - sin(2 * PI* t / float(T));
        ctrlpoints[i][j][0] *= 0.2;
    }
}
t = (t + 1) % T;
glMap2f(GL_MAP2_VERTEX_3,       // Bezier
    0.0, 1.0, 3*VNUM, UNUM,
    0.0, 1.0, 3, VNUM,
    &ctrlpoints[0][0][0]);
glMap2f(GL_MAP2_TEXTURE_COORD_2,    // Texture
        0.0, 1.0, 2, 2,
        0.0, 1.0, 4, 2,
        &texpts[0][0][0]);
glMapGrid2f(60, 0.0, 1.0, 20, 0.0, 1.0);
```

一开始的两次循环是对于控制顶点进行刷新，这是在 loop 的函数里面，每一次都会进行计算，使用的是波动函数的样式。计算好控制顶点之后，就可以进行顶点的计算和纹理的映射了。其中，控制顶点的个数由 UNUM 和 VNUM 表示u,v两个方向的控制顶点数。

2. 在球台上的横幅是用多个四边形面片进行构造的。



比如，如上图的大四边形中，横竖各 4 个顶点，可以用构成 3*3 个小四边形进行组成，横幅中使用的就是这样的思想。

首先，在初始化函数中，我们先将这些顶点的位置计算出来。

```
// initial banner control points
for (int i = 0; i < BANNER_UNUM; i++) {
    for (int j = 0; j < BANNER_VNUM; j++) {
        bannerPoints[i][j][0] = (i * BANNER_LENGTH / BANNER_UNUM) - BANNER_LENGTH / 2;
        bannerPoints[i][j][1] = (j * BANNER_WIDTH / BANNER_VNUM) - BANNER_WIDTH / 2;
        bannerPoints[i][j][2] = 0.5 * sin(((i * 8.0f / 360.0f)*PI*2));
    }
}
```

然后，我们在渲染的时候将这些顶点间的小四边形一一渲染出来，并且贴上相应位置的纹理。详细代码可以见 draw.c:renderBanner()中。

这样构造出的旗帜，它的动画实现比较有意思。类似于物理中的机械波，每一个点的位置都是前一个点在上一时刻的位置。

```
/* Animation */
static int t = 0;
if (t == 10)        // 10 is a period
{
    for (int j = 0; j < BANNER_VNUM; j++) {
        float hold = bannerPoints[BANNER_UNUM - 1][j][2];
        for (int i = BANNER_UNUM - 1; i > 0; i--) {
            bannerPoints[i][j][2] = bannerPoints[i - 1][j][2];
        }
        bannerPoints[0][j][2] = hold;
    }
    t = 0;
}
t++;
```

3. 两类曲面构造的差别：

(1) 对于贝塞尔曲面，只要指出控制顶点的位置之后，既可以构造出曲面。省去了曲面顶点的手动计算，而且纹理一次贴成。其实，对于动画的实现也可以用第二种方法，但是我想尝试一下波动函数的效果，所以用了

两种方法。

(2) 对于小四边形面片组成的曲面，优点是实现起来比较直观，可以直接构造曲面，而不需要通过控制顶点，缺点就是纹理映射有点麻烦，需要更具不同位置进行纹理映射。这在代码中也有所体现。

(二) 场景的构建

在最终版的实现中，我们取消了桌球台的设置，该为天空和大地。

天空是用天空球实现的。首先绘制了一个球体，在其内部贴上天空的纹理。这里球体的法向量是向内的。

```
skyQuadric = gluNewQuadric();
gluQuadricNormals(skyQuadric, GLU_SMOOTH);
gluQuadricOrientation(skyQuadric, GLU_INSIDE);
gluQuadricTexture(skyQuadric, GL_TRUE);
```

地面是读入高度图实现的。高度图是一串 unsigned char 组成的二进制文件，读入内存后我们可以把它当做一个二维数组。在通过两个坐标之后，可以得到第三个高度坐标。当然，高度的范围基本上是在 100-200 之间，所以要经过一系列的坐标平移和放缩。

对于地面，是用一系列的四边形面片组成的，并且通过四个点求得每一个顶点的法向量，供之后的光照。

对于碰撞效果，我们仅仅交换两个球在碰撞方向上的速度。

(三) 光照

这次的光照主要由两种，一种是环境光，另一种是追踪母球的聚光灯。环境光是能够照到全局的，这个光的光源是随着摄像头的移动而移动的。另一种光是用聚光效果（加了聚光角度）实现的，可以追踪母球，通过“G”键进行切换的。

(四) 纹理

对于纹理，有两种实现方式。一种使用二维纹理，直接读入 bmp 文件映射到物体上，另一种是三维纹理，相当于将一个物体浸入一个染缸中。在前者的实现中，有场景的纹理，后者的实现中有球的纹理。

(五) 粒子效果

在球体发生碰撞时，加入了一些粒子效果。所谓粒子，在程序中我们加入了一个结构体，记录他的活跃度、寿命、速度、位置、速度方向和加速度方向等。对于这些小粒子贴上纹理和颜色就可以实现了。

```
// Particles Structure
struct Particles {
    bool    active;                 // Active (Yes/No)
    float   life;                   // Particle Life
    float   fade;                   // Fade Speed
    Point   color;                  // R, G, B
    Point   position;
    Point   velocity;               // velocity direction
    Point   accelarate;             // accelarate direction
};
```