

Quidditch 桌球详细设计说明书

钱泽森 (5130379069)

January 14, 2016

Contents

1 引言	2
1.1 编写目的和范围	2
1.2 术语表	2
1.3 参考资料	2
1.4 使用的文字处理和绘图工具	3
2 技术概要	3
3 模块设计	3
3.1 控制模块	3
3.1.1 沙盒 (Arena)	3
3.1.2 控制器 (Controller)	3
3.1.3 球 (Ball)	4
3.1.4 幽灵球 (GhostBall)	4
3.1.5 母球 (CueBall)	4
3.1.6 游走球 (WanderBall)	4
3.1.7 金色飞贼 (SnitchBall)	5
3.1.8 迷球 (FantasyBall)	5
3.1.9 桌面 (Ground)	5
3.2 粒子系统	5
3.2.1 火花 (Spark)	6
3.2.2 烟雾 (Smoke)	6
3.3 软体系统	7
3.3.1 布料 (Cloth)	7
3.4 绘图模块	7
3.4.1 场景 (Scene)	7
3.4.2 可渲染物体 (Render)	7
3.4.3 形状 (Shape)	8

3.4.4	光照 (Light)	10
3.4.5	粒子系统 (Particle)	11
3.4.6	视角 (View)	11
3.4.7	投影 (Projection)	11
3.5	音效系统	12

1 引言

1.1 编写目的和范围

本详细设计说明书编写目的是说明程序模块的设计考虑，包括程序描述、输入/输出、算法和流程逻辑等，为软件编程和系统维护提供基础。本说明书的预期读者为系统设计人员、软件开发人员、软件测试人员和项目评审人员。

1.2 术语表

Buffer Object An object that represents a linear array of memory, which is stored in the GPU. There are numerous ways to have the GPU access data in a buffer object.

Context, OpenGL A collection of state, memory and resources. Required to do any OpenGL operation.

OpenGL Shading Language The language for writing Shaders in OpenGL.

Shader A program, written in the OpenGL Shader Language, intended to run within OpenGL.

Texture An OpenGL object that contains one or more images, all of which are stored in the same Image Format.

OpenGL A cross-platform graphics system with an openly available specification.

1.3 参考资料

资料名称	作者	文件编号/版本	资料存放地点
OpenGL Tutorial	Unknown	latest	http://www.opengl-tutorial.org
OpenGL step by step	Unknown	Latest	http://ogldev.atspace.co.uk/
OpenGL wiki	collaborators	Latest	https://www.opengl.org/wiki/
OpenGL 3.3 Reference Pages	SGI	3.3	https://www.opengl.org/sdk/docs
Bullet Physics Doc	Bullet	2.83	http://bulletphysics.org/Bullet

1.4 使用的文字处理和绘图工具

文字处理软件 Emacs, Org Mode

UML 图生成 Doxygen

2 技术概要

- 基于 OpenGL 3.3 core API
- 使用了 SFML 作为 window system
- 使用 glew 作为 Extension Wrangler Library
- 使用 glm 作为数学库
- 使用 Bullet Physics 作为物理引擎
- 开发环境为 Linux x86-64 + Mesa

3 模块设计

3.1 控制模块

3.1.1 沙盒 (Arena)

沙盒模块负责对物理世界进行仿真模拟. 由于本软件已经使用 Bullet Physics 作为物理引擎, 因此本软件的着重点在于将 Bullet Physics 应用到程序中. Arena 实际上是对 Bullet 的一个包装. 我们定义了如下几个类.

3.1.2 控制器 (Controller)

这是一个抽象类, 用来控制游戏世界中的一个 (SingleController) 或者多个 (GroupController) 实体. 本类中定义了一个虚方法:

```
virtual bool control(const float elapsed);
```

所有继承类都要实现这个虚方法. 在每个时间片的开始, Arena 都会调用这个方法, elapsed 表示距离上次调用已经过去了多少时间. Controller 需要根据自己的用途, 来控制自己所代表的刚体. 这个控制, 既可以是来自用户的 (CueBall), 也可以是来自自身的 (WanderBall). 下面对 Controller 最主要的几个子类进行解释. 继承关系请见图1.

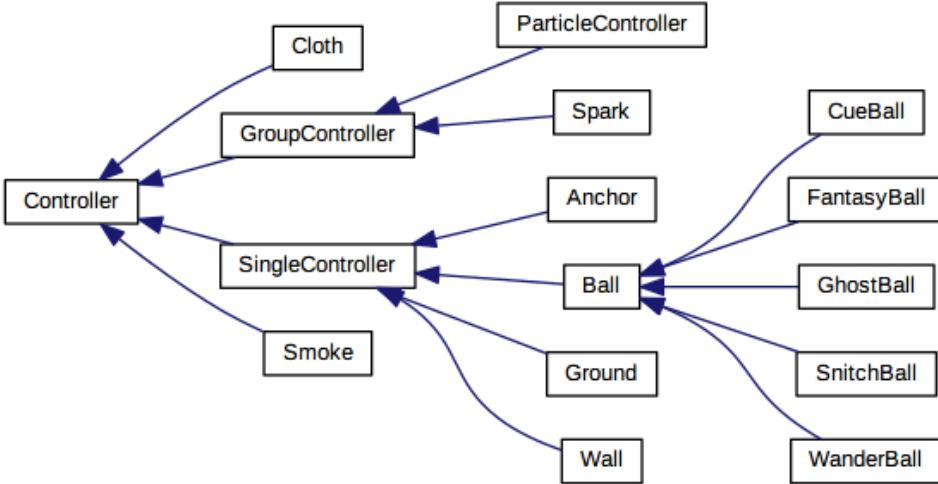


Figure 1: Controller 的继承关系

3.1.3 球 (Ball)

所有球类的父类. 主要用于归类, 没有特殊用途.

3.1.4 幽灵球 (GhostBall)

普通球, 没有任何行为, 因此他的 control 方法什么也不做.

3.1.5 母球 (CueBall)

受用户控制, 每个时间片都会读取用户的键盘输入, 并且对所控制的刚体球施加一个力.

3.1.6 游走球 (WanderBall)

自主随机游走的球, 每个时间片都会检查当前的速度和理想速度, 并且逐渐趋近这个理想速度.

3.1.7 金色飞贼 (SnitchBall)

会飞的球, 每隔一段时间都会离地飞行, 一段时间后又会落回地面.

3.1.8 迷球 (FantasyBall)

我为了增加游戏性, 特意增加的一种球, 作为一种惩罚措施. 母球碰到该球后, 之后一段时间的操作都会被反向地执行.

3.1.9 桌面 (Ground)

指的是小球运动的桌面. 凹凸不平的地面是由 Perlin 函数生成的.

3.2 粒子系统

为了增加真实性, 我对粒子系统做了如下处理:

- 每个粒子都是一个正方形的 billboard, 也就是说, 我们通过一些计算, 使得该正方形的面始终正对摄像机镜头. 这样做有几个好处:
 - 减少了一半绘画的面数 (否则至少需要一个四面体来保证从各个方向都能看到微粒)
 - 效果更加真实 (四面体的微粒不真实).

```
fragPos = centerPos  
+ cameraRight * vert.x * size  
+ cameraUp * vert.y * size;
```

- 锋利的边缘让微粒看起来很不真实, 因此我在 Fragment Shader 中做了处理: 对于同一个微粒, 正中央的 alpha 最高, 边缘的 alpha 为零, 中间平滑过渡. 这样子的微粒才有朦胧的感觉. 效果对比请见图2.

```
surfaceColor.a *= 1 - distance(fragPos, centerPos) / size;
```

为了提高性能, 我做了如下优化:

- 使用了 OpenGL 提供的 Instancing 接口. 对于数万个微粒, 如果对于每个微粒都调用一次 OpenGL 的绘图函数, Overhead 太大, 无法进行实时的流畅绘画. 因此我采用了 OpenGL 的 Instancing, 对于数万个微粒, 只需要提供单个微粒的样板 (如上所说, 是一个正方形) 和每个微粒的位置, 只需要一次 OpenGL 的函数调用, 就可以画出数万个微粒, 极大地提高了性能.

```
glDrawArraysInstanced(GL_TRIANGLE_STRIP, 0, 4, vertOffset.size());
```

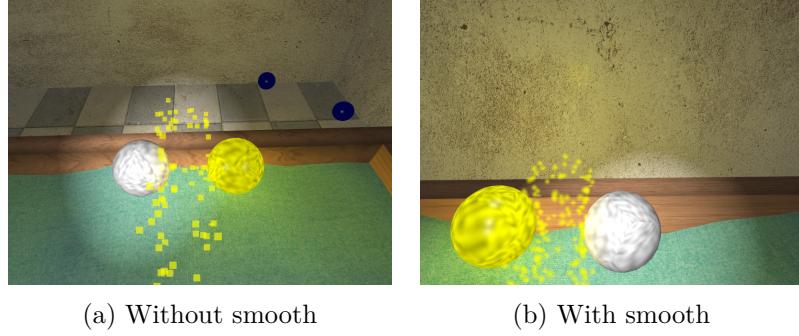


Figure 2: 开启平滑前后对比

3.2.1 火花 (Spark)

在白色母球和金色飞贼碰撞时触发. 包含数千个微粒, 在这里我使用了 Bullet 来模拟火花的行为, 理由如下:

- 火花会收到重力的影响, 应该放到物理系统里模拟.
- 火花会和别的物体碰撞, 这一点也应该放到物理系统来模拟. 一个意外的收获是, 只要每个火花的质量不太小, 火花能够对周围的物体产生可见的影响, 我利用这一点, 使得白色母球和金色飞贼在碰撞后会像触电一样弹开.

另外我做了一个小细节, 使得每个时间周期都会对每个火花的颜色做调整 (逐渐变暗), 使得火花在整个生命周期中显现出一种“逐渐熄灭”的感觉.

3.2.2 烟雾 (Smoke)

从迷球表面散发, 用来警告用户. 包含数万个微粒.

为了追求真实性, 在离子系统的通用优化的基础上, 我进一步做了如下处理:

- 每个微粒都有一个生命周期, 在这个生命周期内, 他的 alpha(不透明度) 会逐渐减小直到变成零, 然后被删除. 这种做法比较好的模拟了烟雾逐渐消散的过程. 效果对比请见图3.

为了提高性能, 在粒子系统的通用优化的基础上, 我做了如下优化:

- 不使用 Bullet Physics 的物理系统来模拟微粒的运动, 而使用噪声函数来仿真. 为了真实性, 我们的微粒数量应该尽可能多, 以数万为最佳. 经过测试, Bullet 在这种数量级的物体下, 无法进行实时的流畅模拟.

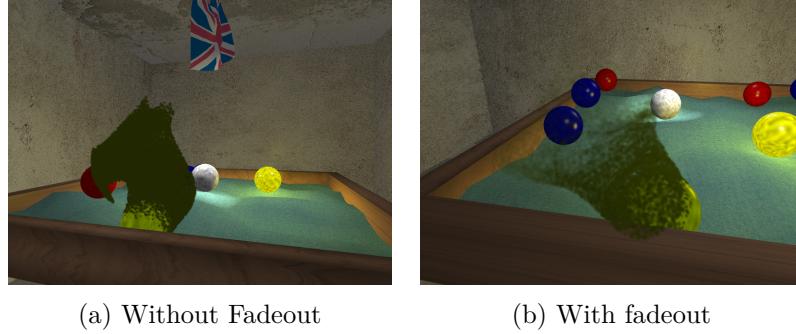


Figure 3: 开启渐变前后对比

因此我放弃了纯物理的方式, 转而投向了噪声函数模拟. 在这里我们复用了 Perlin 函数, 来生成每个微粒在下一个时间片的速度. 噪声的参数是绝对时间和小球的绝对方位. 这样可以保证烟雾在时间和空间上都保持连续, 但是都保持逐渐变化.

3.3 软体系统

3.3.1 布料 (Cloth)

本质上是一个长方形的 Mesh, 包含数百个小三角形. 相邻的几个节点之间会有 link 相连, 来限制他们的相对运动. 关于风的模拟: 在每一个时间片中, 对于该软体的每个节点, 都从一个噪声函数生成一个受力. 这个噪声函数的参数是该节点的绝对位置和当前绝对时间. 这样既保证了受力在时间和空间上的连续性, 又保证了一定的变化.

3.4 绘图模块

3.4.1 场景 (Scene)

表示一个完整的场景, 包括一些 Render, Light, Particle.

3.4.2 可渲染物体 (Render)

是一个抽象类, 只是定义了一个通用接口, 调用后即会画出该物体, 所有的继承类都要实现该接口.

```
virtual void render(ModelSetter ms, MaterialSetter ts) const = 0;
```

`ModelSetter` 是一个回调函数, 来设定这个物体在全局中位置. `MaterialSetter` 也是一个回调函数, 来设定这个物体的材料. 材料的定义包括:

- 纹理
- 反光度
- 反光的颜色
- 自发光亮度

继承关系请见图4.

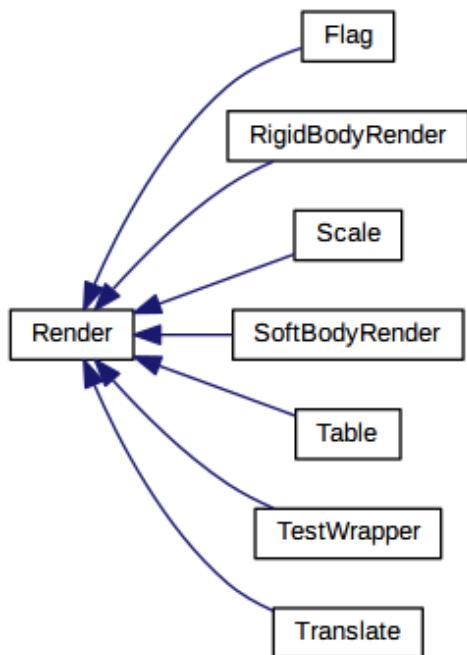


Figure 4: Render 的继承关系

3.4.3 形状 (Shape)

也是一个抽象类, 定义了一个通用接口, 调用后即会画出该物体.

```
virtual void render(Render::ModelSetter ms) const = 0;
```

其中 ModelSetter 是一个回调函数, 来设定这个物体在全局中位置. 继承关系请见图5.

下面着重讲一下 Sphere (球) 的实现方法.

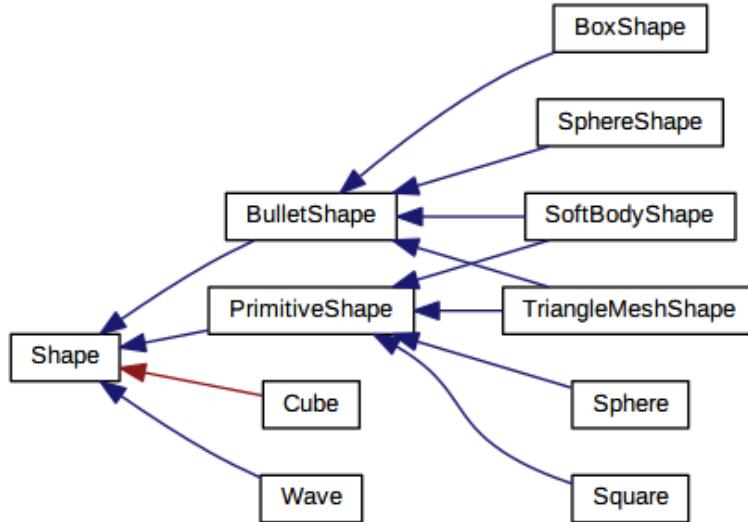


Figure 5: Shape 的继承关系

1. 球 (Sphere)

大多数人都会选择使用 `gluSphere` 函数 (见图6) 来直接生成球体, 但是我认为这种方法有如下缺陷:

- 划分不均匀, 在两极点的划分明显要比赤道的划分密得多. 为了达到某个划分细度, 则必须要在赤道达到该密度, 则此时极点的密度是过高的, 造成了绘图资源的浪费.
- 难以复用. 对于每一个球体, 用户都需要调用 `gluSphere` 来绘出这一个球体. 这至少造成了两方面的浪费. 一是每次调用都需要重新计算每个节点的坐标, 造成了 CPU 资源的浪费. 二是每个球体都需要把节点坐标传到 GPU 中, 造成了 GPU 带宽和 GPU Memory 的浪费.

由于这些原因, 我没有采用这种简单的办法, 而是用三角形剖分的方式来生成球形 (见图7). 方法如下:

- (a) 首先生成一个正四面体.
- (b) 对于四面体的每个三角面, 均匀分成四个三角面.
- (c) 将新生成的节点, 延长到球体的表面上.
- (d) 重复 2,3 步骤, 直到足够的精度为止.

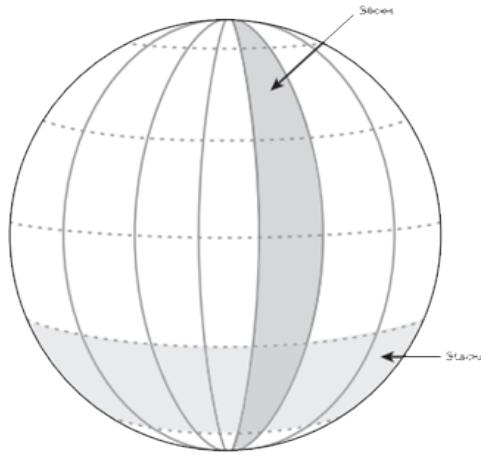


Figure 6: gluSphere 的划分方式

实际使用过程中, 我大概需要迭代 3 次, 也就是说, 大约 256 个面的球体, 已经非常细腻了. 这一方法完美解决了上面提到的问题.

3.4.4 光照 (Light)

在 Phong 的光照模型上, 做了几个改进:

- 加了自发光系数, 来保证发光物体看起来是亮的.
- 加了衰减系数, 使得较远的物体看起来较暗.
- 加了光照方向, 来支持聚光灯.

```
struct Spec {
    glm::vec4 position; //光源位置
    glm::vec3 intensities; //光色
    float attenuation; //衰减系数
    float ambientCoefficient; //环境光照系数
    glm::vec3 coneDirection; //光照方向
    float coneAngle; //方向角宽
};
```

继承关系请见图8.

FollowSpotlight 也就是聚光灯, 会跟随某个物体移动.

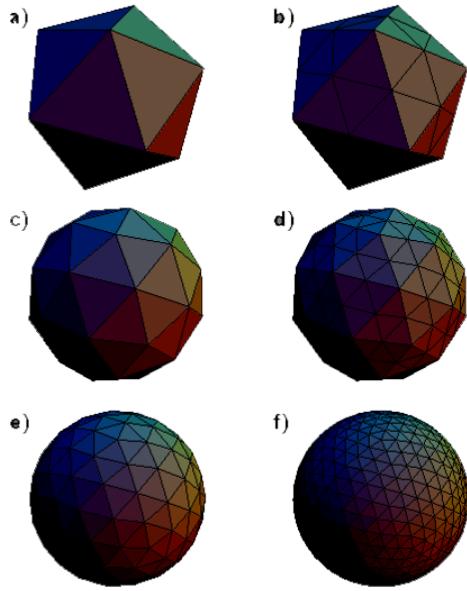


Figure 7: 更好的划分方式

MovingLight 依附在某个物体上的灯, 主要用来支持自发光物体, 比如金色飞贼

SimpleLight 最简单的的灯, 所有系数都是固定的.

ToggleLight 支持开关的灯.

3.4.5 粒子系统 (Particle)

请见3.2.

3.4.6 视角 (View)

表示用户的视角, 包括摄像头所在的位置, 观察的方向等等, 可以调用获取对应的 View Matirx.

3.4.7 投影 (Projection)

表示摄像头的投影, 包括横向的和纵向的视角, 最近的切点和最远处的切点, 可以调用获取对应的 Projection Matrix.

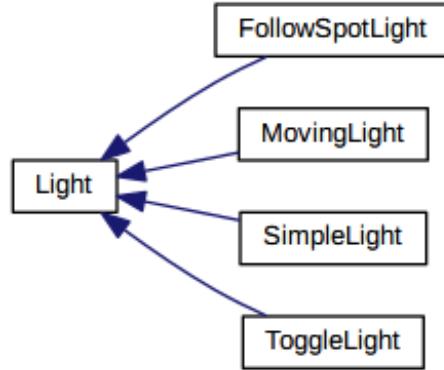


Figure 8: Light 的继承关系

3.5 音效系统

我实现了一个立体声的音效系统, 实现方法如下:

- 维护一个向量来表示玩家视角当前的位置, 一个向量表示玩家当前的朝向.
- 对于场景中产生的某次碰撞或其他事件, 根据多个因素, 来决定双声道中音量的大小. 总的来说, 具体的音效播放与下列因素均有关系:
 - 玩家所在的位置, 和面对的方向
 - 音效所在的位置
 - 该事件的烈度, 比如碰撞时产生的弹力大小

目前我对球与球之间的碰撞, 以及球与挡板之间的碰撞, 均做了碰撞的音效.