Quidditch 桌球详细设计说明书

钱泽森 (5130379069)

December 17, 2015

Contents

1 引言

1.1 编写目的和范围

本详细设计说明书编写的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	collabrators	Latest	`https://www.opengl.org/wiki/`
OpenGL 3.3 Reference Pages	SGI	3.3	`https://www.opengl.org/sdk/docs`

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
- 开发环境为 Linux x86-64 + Mesa

3 模块设计

3.1 用例图

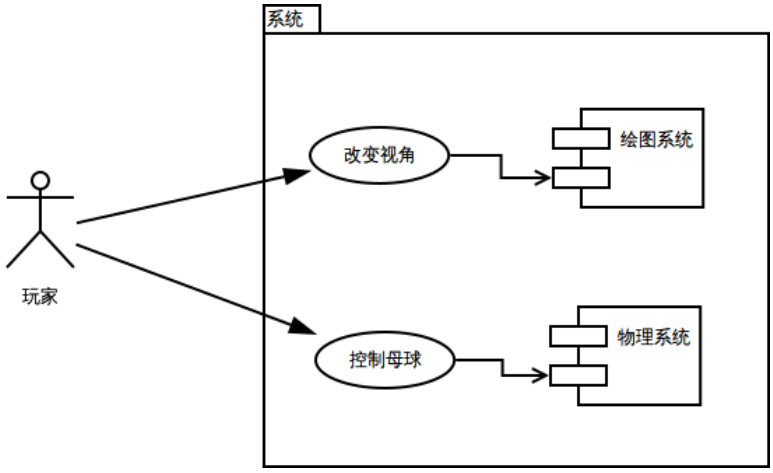

Figure 1: 用例图

3.2 功能设计说明

3.2.1 物理模块

物理模块主要由沙盒 (Arena) 和各个物理元件 (SnitchBall, GhostBall, CueBall, WanderBall, Wall) 构成.

1. 沙盒 (Arena) 沙盒模块负责对物理世界进行仿真模拟, 它负责的物理模拟主要有以下几类:

滚动摩擦 在桌面上的小球在不受到其他外力的情况下, 会收到桌布的滚动摩擦力, 因此速度会越来越小.

小球间碰撞 小球之间会产生非弹性碰撞, 碰撞后两球的速度和方向都会发生变化, 此过程会有能量损失.

小球和桌面以及墙面的碰撞 小球和桌面以及墙面也会产生非弹性碰撞.

沙盒每次都往前演绎一段时间, 并且不断修改球桌上元件的物理属性, 来反映物理世界的规律.

2. 球 (Ball) 所有球类的父类, 主要记录球的质量/半径/位置/速度. Ball 的继承关系请见 2.

3. 幽灵球 (GhostBall) 普通球, 直接继承自 Ball, 没有额外的属性.

4. 母球 (CueBall) 用户可以操作的球, 继承自 GhostBall, 没有额外的属性.

5. 游走球 (WanderBall) 自主随机游走的球, 继承自 Ball. 额外属性: v0 表示理想速度, mu 表示趋近速度. 在当前速度不等于理想速度时, 本球会以 mu 的速率逐渐趋近于 v0.

6. 金色飞贼 (SnitchBall) 会飞的球. 继承自 Ball, 需要额外记录当前状态下剩余的时间 (以秒表示), 以及理想的飞行速度.

7. 墙 (Wall) 表示沙盒的边缘, 既可以用来表示垂直的墙, 也可以表示地面或者天花板. 属性有: 一次方程的一组参数, 用来表示墙的平面. 以及一个弹性系数, 用来表示球和墙碰撞后的能量损失程度.

3.2.2 绘图模块

绘图模块主要负责在屏幕上绘图.

1. 可渲染物体 (Renderable) 定义了一个通用接口, 调用后即会画出该物体.

2. 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. 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.

4. 场景 (Scene) 表示一个完整的场景, 包括一些 Renderable, 一个 View, 一个 Projection. 调用后即会画出整个场景.

5. 形状 (Shape) 表示一个固定的形状, 比如, 处在原点且半径为 1 的球, 或者处在原点且边长为 1 的立方体, 等等.

6. 旗帜 (Flag) `Flag` 实际上是 `Wave` 的一个包装, 后者才是真正的飘动动画. Wave 用了两种实现, 一种是三角函数 (`WAVE_TRIANGLE`), 一种是 Bezier 函数 (`WAVE_BEZIER`).

   `WAVE_TRIANGLE` 的实现比较简单, 主要的难点在于旗帜的前后波动. 对于一个旗帜上的一个点, 这个点的深度就是一个三角函数函数, 其参数与下列因素有关:

   - 这个点在旗帜上的位置 (横向, 纵向)
   - 当前的时间

   根据这些算法就能实现旗帜飘动的效果.

   `WAVE_BEZIER` 的实现相对比较复杂, 首先写了一个类 `Bezier`, 用来进行 Bezier 曲线相关的计算. 计算步骤如下:

   (a) 对于每个 control point, 生成一个随机的频率 f. 这个频率这个 Wave 的生命周期中不会改变.

   (b) 对于每一帧画面, 用 $sin(f * t)$ 作为这个 control point 的深度.

   (c) 根据这些 control point, 生成整个曲面.

   按照这种思路, 实际运行的时候发现有些卡顿, 为此我针对我的游戏实际的需求, 对曲面计算做了一些优化:

   (a) 对于 Bezier 曲线公式中的组合运算, 开启一块 cache, 这样对于同样的 n 和 m, $C(n, m)$ 只需要计算一次.

   (b) 考虑到曲线计算的采样点实际上是固定的, 因此先用这组固定的采样点计算好 Bezier 公式中的一大部分, 以后的每次调用只需要 O(nm) 的复杂度. 而不是 $O(nmWH)$ 的复杂度.

   加了上述两个简单的优化后, 动画基本没有卡顿了.

   `Flag` 再在此基础上加上纹理 (也就是旗帜图案). 这样, 一面冉冉飘动的旗帜就做好了.

   目前的缺点在于, 对于每一帧动画, 都需要重新传入顶点数据, 可能对性能会有一定影响 (虽然实际中没有遇到卡顿). 目前正在寻找更好的方案.

3.2.3 接口模块

接口模块主要用来衔接物理模块和绘图模块.

1. BallWrapper 一个包装类, 是 Renderable 的子类. 主要用来将 Ball 的物理属性, 转化为 Renderable 的绘图属性. 比如, 如果 Ball 的 x 属性是 (1, 0, 0), 则生成的 Renderable 会在 (1, 0, 0) 处画一个球.
2. Table 一个包装类, 是 Renderable 的子类. 主要用来将 Arena 的物理属性, 转化为 renderable 的绘图属性.

3.2.4 控制模块

负责读取用户的输入并且修改对应的游戏状态. 比如鼠标移动, 则修改 View 的相关属性, 等等.

4 接口设计

4.1 内部接口

4.1.1 Shape

`virtual void draw() const`
调用即绘出这个形状.

4.1.2 Renderable

`virtual void render(const GLuint WVP, const glm::mat4 &mat) const`

WVP World-View-Projection 矩阵所对应的 Uniform Variable 编号

mat 已经算出的 View-Projection 矩阵

调用后, Renderable 会计算出最终的 WVP 并且绑定, 最终绘出该物体.

4.1.3 Arena

`void deduce(const float t)`

t 距离上次调用逝去的时间, 单位为秒

调用后, Arena 会推算物体运动并且修改响应的物体的状态.

`void attach(Ball *ball)`

ball 需要添加的球

`void attach(Wall *wall)`

wall 需要添加的墙

4.1.4 Scene

`Scene(const View &view, const Projection &projection)`

view 观察该场景的视角

projection 观察该场景的投影

`void render()`
调用即绘出这个场景.
`void attach(Renderable * renderable)`

renderable 需要添加的物体

5 系统性能设计

为了性能考虑, 本软件使用 OpenGL 3.3 Core API, 避免了大量的 `glBegin()` 和 `glEnd()` 调用, 极大地提高了性能.

6 系统出错处理

本系统可能遇到的错误有以下几类

载入纹理错误 致命错误, 程序直接退出

载入音效错误 致命错误, 程序直接退出

其他错误 输出到终端, 不影响程序运行

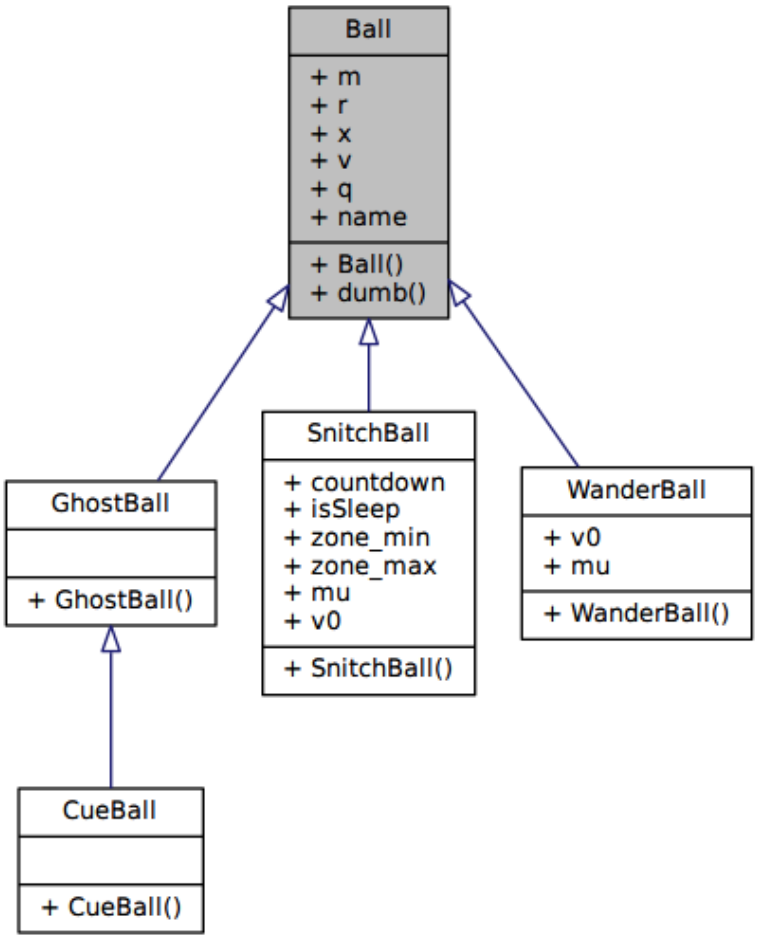

Figure 2: Ball 的继承关系