简单魁地奇 Simple Quiditch

游戏规则

本游戏采用闯关的形式，以白球落入黑袋作为一关的完成。在白球发球后，玩家无法操控白球，只能通过旋转桌面，改变所有球的反弹轨迹，最终反弹入袋。在桌面上有 6 个静止球和 6 个游走球作为障碍物，妨碍玩家的白球落入黑袋。还有一个金色飞贼在球桌上空随机运动，当金色飞贼掉落到桌面后，如果白球成功击中金色飞贼，可以获得大量分数。

积分规则

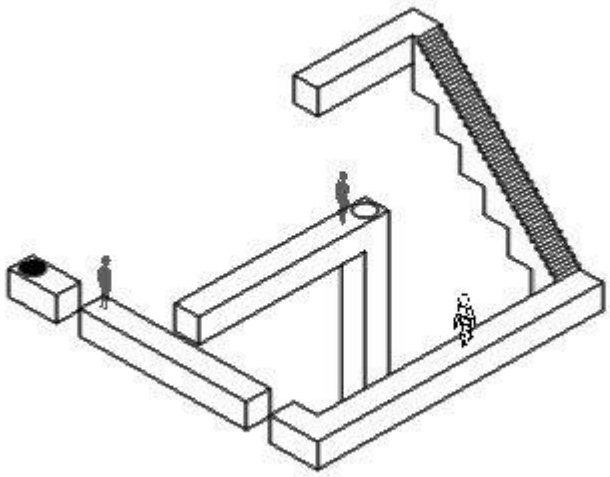
在每一关中，随着时间的流逝，会导致分数的减少，每一秒扣除 5 分。每次击中金色飞贼可以获得 100 分。分数实时显示在屏幕上，并在闯关成功后结算。

操控方法

在游戏开始后可以按 A 和 D 来旋转桌面，然后可以按 Space 来打击白球开始运动。

美术风格

本游戏借鉴《无限回廊》的美术风格，以黑白色调为主，由于必须体现技术上的实现，因此添加了光照系统。最终准备采用非真实渲染的方式，呈现整个游戏。



无限回廊游戏截图

已实现功能

1. 白球、静止球、游走球和金色飞贼的模型建立。

2. 桌面的模型建立。
3. 白球、静止球的运动（平动）动画。
4. 金色飞贼的随机运动动画。
5. 球和球之间的碰撞检测。（对于静止球未开启碰撞功能，详见 collidable 变量）
6. 球和桌面之间的碰撞检测。
7. 镜头的旋转。
8. 键盘事件的记录。
9. 垂直同步功能。
10. 光源的设置。
11. 球体、桌面的材质设置。
12. 分数系统。
13. 提示信息绘制。

对应作业要求

1. 设计搭建桌球模型（15 分）

   完成，详见 Desk 类。
2. 桌面放置 6 个静止的“鬼球”，初始状态是静止（后续作业中鬼球在受到其他球碰撞时会滚动并由于摩擦力停止）（10 分）

   完成，详见 StaticBall 类。
3. 桌面还有 6 个任意缓慢移动的“游走球”，沿任意路径缓慢滚动，（后续作业中游走球会在缓慢滚动时避让其他球，只有在被撞击速度加快时会撞到其他的球）；（20 分）

   完成，详见 MovingBall 类。
4. 1 个“金色飞贼”小球，桌面上空随机飞行，但会间歇性落到平面上休眠，在游戏中击中该球会压倒性地取得高分；（20 分）

   完成，详见 GhostBall 类。
5. 1 个白色母球，受球杆打击会沿球杆运动方向以某种速度滚动，并撞击其他球；（15 分）

   完成，详见 WhiteBall 类，球杆的打击以完成碰撞系统呈现，初始白球速度由咒语发动。
6. 设计游戏模式和积分规则；（10 分）

   完成，此文档。
7. 详细的设计报告。（10 分）

   完成，此文档。

整体架构

游戏由多个类进行管理，在设计上尽量向 Unity 学习（下次迭代会更新架构），以下是详细介绍。

GameManager 类

这个类为整个游戏的管理类，其中存储了所有关卡和当前关卡，并在渲染和碰撞检测的时候负责调用渲染器和碰撞器。

对于游戏，一共有三个阶段，首先是初始化阶段，这个阶段负责读入关卡和 OpenGL 开关选项设置；其次是计算阶段，这个阶段分为键盘事件处理（键盘事件不立即处理而是延时到下一次的计算阶段处理）、碰撞处理、运动管理和分数计算；最后是绘制阶段，这个阶段分为绘制初始化（负责光影初始化和镜头初始化）和模型绘制。

在所有处理和渲染过程中，渲染相关的调用渲染器、碰撞相关的调用碰撞器、镜头相关的调用镜头类以及运动相关的调用该关卡的每个游戏对象。

Renderer 类

这个类负责整个游戏的渲染，为纯静态方法类，通过对 Render 函数不同游戏对象类型参数的重载实现对不同对象的不同渲染方法。对于桌面和球体，绘制的过程分为两步骤，一个是绘制填充实体，另一个是绘制外框（为了实现非真实渲染）。

Physics 类

这个类负责整个游戏的碰撞检测，为纯静态方法类，通过对 CollisionDetect 函数不同游戏对象类型参数的重载实现对不同对象的不同碰撞方法。无论是球对桌面的碰撞还是球对球的碰撞，都分为两个步骤，一是速度计算，二是位置微调。在速度计算时，球对桌面就是简单的镜面反射，球对球则是法线方向速度交换、切线方向速度不变的方法；在位置微调时，球对桌面的情况会调整球和桌面不重叠，球对球的情况会根据计算后的速度方向运动直至两球不重叠，这样可以较好地避免球和球之间一直重叠的情况。

Camera 类和 Lighter 类

这两个类负责整个游戏的光影和镜头，为纯静态方法类，主要是 OpenGL 的一些设置，不作赘述。

Level 类

这个类负责管理每个关卡的游戏对象，包括桌面、球体、黑袋和文本，并有一个 move 方法来使每个对象进行运动。

Ball 类和 BallGUI 类

Ball 类表示每一个球体，其中有一个 BallGUI 类实例来表示这个球体的材质，另外还记录了球的位置、速度和可碰撞性等。

一共有四个类继承于这个类，它们是 WhiteBall, GhostBall, MovingBall 和 StaticBall 类，它们分别有各自的构造方式和不同的运动方式。

Desk 类

这个类表示一个桌面，包括桌面图形上包含的立方体和物理上的碰撞边界。

Edge 类和 Rect 类

这两个类主要表示上述物理上的碰撞边界和图形上的立方体，是比较单纯的数据结构体。

Text 类

这个类用来表示一个文本，包括内容和位置。

Point 类和 Vec 类

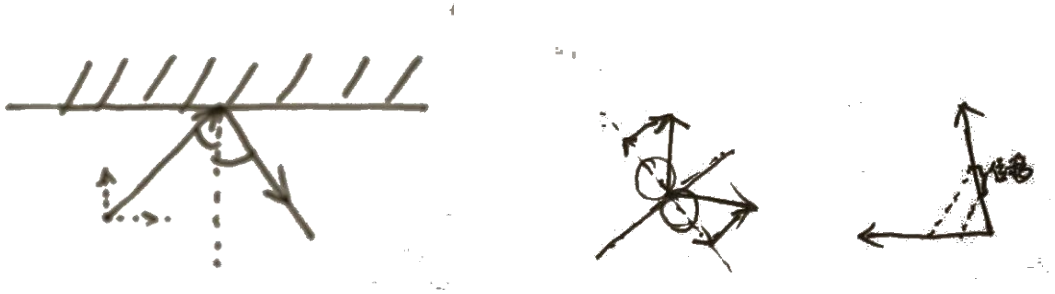
这两个类分别表示三维空间中的一个点和一个向量，和实际的物理意义作用相同。

详细算法

碰撞

球与面碰撞：

首先分解球相对于碰撞面的速度法向量和切向量。接着将法向量交换，最后将原先的切向量和新的法向量合成，形成新的速度。为了防止球陷入面内，会将球的位置按照法线方向移动至面外紧贴面的地方，从而解决球与面相交问题。



球与面碰撞模型和球与球碰撞模型

球与球碰撞：

首先分解两球相撞的速度，也是按照相撞点的切向方向和法向方向，接着交换两球的法向方向，最后将原先的切向量和新的法向量合成，形成新的速度。为了防止两球相交，将两球分别沿新速度方向运动，直至两球之间距离等于两球半径和，从而解决球与球相交问题。

相关代码如下：

```
void Physics::collisionDetection(const Edge& edge, Ball& ball) {
    if (edge.direction == true) { // x direction
        if (abs(ball.Pos().y - edge.start.y) <= ballRadius) {
            ball.flipY();
        }
    }
    else {
        if (abs(ball.Pos().x - edge.start.x) <= ballRadius) {
            ball.flipX();
        }
    }
}

void Physics::collisionDetection(Ball& ball_1, Ball& ball_2) {
    if (!(ball_1.collidable && ball_2.collidable))
        return;
    auto disVec = ball_1.Pos() - ball_2.Pos();
    auto dis = disVec.length();
    if (dis <= 2 * ballRadius) {
        auto vertical = disVec / dis;
        auto horizontal = Vec(vertical.y, -vertical.x, vertical.z);
        auto v1 = (ball_2.velocity * vertical) * vertical + (ball_1.velocity *
horizontal) * horizontal;
        auto v2 = (ball_1.velocity * vertical) * vertical + (ball_2.velocity *
horizontal) * horizontal;
        auto v = v1 - v2;
        auto scale = (2 * ballRadius - dis) / v.length();
        ball_1.pos = ball_1.pos + (scale * v1);
        ball_2.pos = ball_2.pos + (scale * v2);
        ball_1.velocity = std::move(v1);
        ball_2.velocity = std::move(v2);

    }
}
```

渲染

对于非真实性渲染，有一点比较明显的特点就是模型轮廓线的渲染。目前没有找到很好的解决方案，只能用一种比较简陋的方法实现，就是填充模型和线框模型的结合。由于线框也会被光照所影响，所以一个模型的绘制过程是这样的：

打开光照->绘制填充模型->关闭光照->绘制线框模型

相关代码如下：

```
    glEnable(GL_LIGHTING);
    Renderer::renderFill(currentLevel.desk);
    glDisable(GL_LIGHTING);
    Renderer::renderOutline(currentLevel.desk);

    for each (auto ball in currentLevel.balls) {
        glEnable(GL_LIGHTING);
        Renderer::renderFill(*ball);
        glDisable(GL_LIGHTING);
        Renderer::renderOutline(*ball);
    }
```

下一次迭代会试着使用 GLSL 来实现片段式阴影，希望助教能够知道如何渲染轮廓线。