计算机图形学大作业设计报告

简单魁地奇 Simple Quiditch

目录_Toc440321031

蓝色部分为迭代三修改部分

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

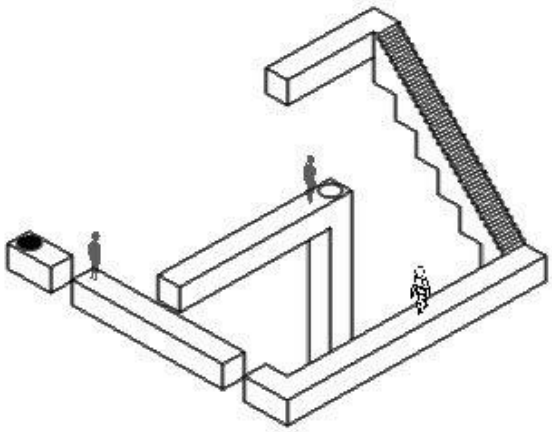
在每一关中，随着时间的流逝，会导致分数的减少，每一秒扣除 5 分。每次击中金色飞贼可以获得 100 分。分数实时显示在屏幕上，并在闯关成功后结算。

操控方法

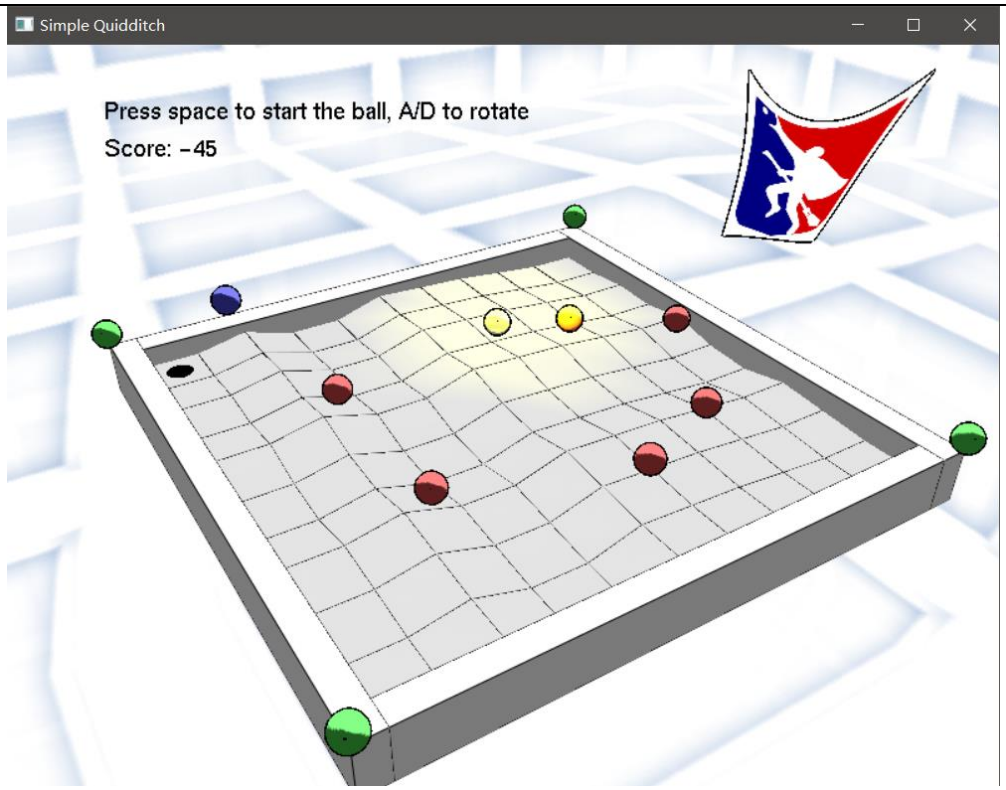
在游戏开始后可以按 A 和 D 来旋转桌面，然后可以按 Space 来打击白球开始运动，按 W 可以打开关闭聚光灯效果。

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



Simple Quiditch 游戏截图

已实现功能

1~13 为第一次迭代完成功能，14~17 为第二次迭代完成功能

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。
14. 旗帜建模与绘制
15. 旗帜布料物理模拟

16. 旗帜材质导入为魁地奇赛事旗帜
17. 完成非真实性渲染
18. 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
19. 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）
20. 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）
21. 使用 perlin 噪声函数添加自然纹理到小球表面上
22. 球体滚动效果
23. 桌面转动的动态模糊效果
24. 游戏逻辑（包括游戏开始和游戏终止）
25. 游戏音效播放

对应作业要求

1~6 为第一次迭代作业要求，6~10 为第二次迭代作业要求

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. 用各种曲线函数设计并构建旗帜模型（35 分）
   完成，详见 Flag 类，后有详细解释。
8. 在旗帜模型上加入自己独特的贴图（20 分）
   完成，详见 Renderer 类，后有详细解释。
9. 在旗帜模型中加入旗帜飘扬的相关动画（30 分）
   完成，详见 Flag 类，后有详细解释。

10. 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）；

    完成，详见 Tile 类。

11. 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）；

    完成，详见 particle 和 emitter 类。

12. 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）；

    完成，详见 lighter 类。

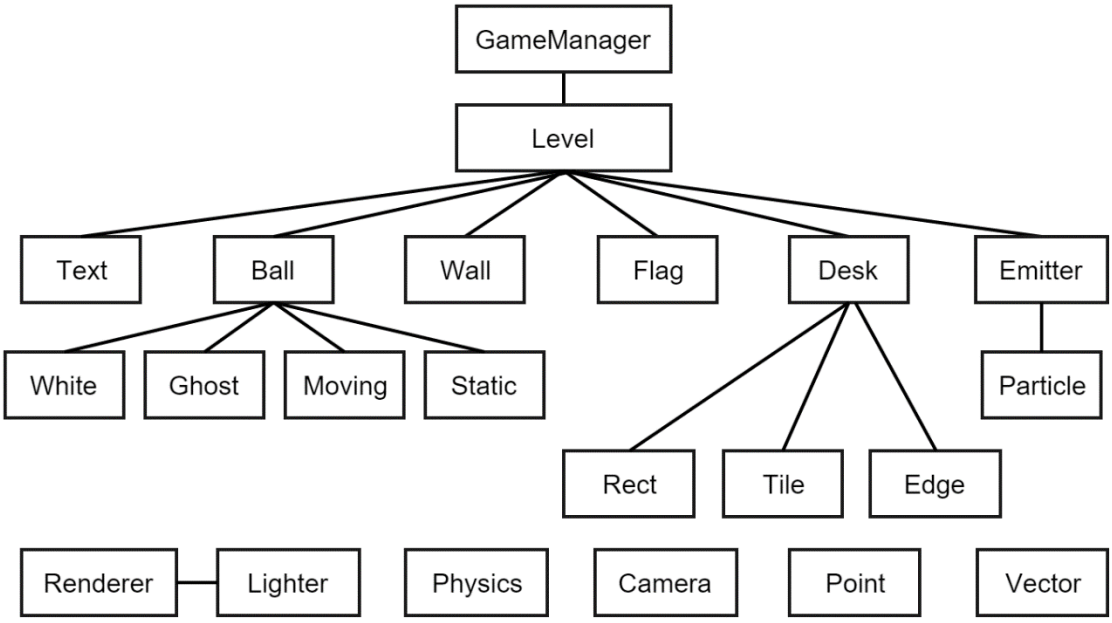
13. 纹理：使用 perlin 噪声函数添加自然纹理到小球表面上；（25 分）

    完成，详见 whiteball 类，后有详细解释

14. 详细的设计报告以及标准格式的提交文件（15 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

```
// INIT part
// Some initialization work
void init();
// API for level load in the future
void loadLevel(int level = 0);

// DISPLAY part
// Some initialization of rendering every frame
void displayInit();
// Call renderer to render
void render();

// IDLE part
// Call objects to move
void move();
// Call physics module to calculate
void physics();

// KEYBOARD part
// Reshape the window
void reshape(GLfloat width, GLfloat height);
// Record key press and call according events
void keyEvent();

// LOGIC part
// Calculate score
void score();
```

这个类为整个游戏的管理类，是一个单例，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。

Physics 类

```
class Physics {
public:
    static bool collisionDetection(const Edge& edge, Ball& ball);
    static void collisionDetection(Ball& ball_1, Ball& ball_2);
    static void collisionDetection(const Desk& desk, Ball& ball);
    static bool overlapDetection(const Ball& ball_1, const Ball& ball_2);
    static void positionAmend(Ball& ball, const Desk& desk);
    static bool heightAmend(Ball& ball, const Desk& desk, bool down);
    static void gravityApply(Ball& ball, const Desk& desk);
    static void normalCalc(Ball& ball, const Desk& desk);
};
```

这个类负责整个游戏的物理模型，包括碰撞检测，重叠修复和重力影响三个部分，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。在重叠位置微调时**，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。详细的算法在后文中有阐述。

Renderer 类

```
class Renderer {
    static void renderOutline(const Desk& desk);
    static void renderFill(const Desk& desk);
public:
    static void init();
    static void displayInit();
    static void render(const Desk& desk);
    static void render(const Text& text);
    static void render(Ball& ball);
    static void render(const Point& point);
    static void render(int score);
    static void render(Flag& flag);
    static void render(const Wall& wall);
    static void render(const Emitter& emitter);
    static void render(const Particle& particle);
    static void render(const Tile& tile);
};
```

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）。在第二次迭代中，我优化了 Renderer 类的接口，使得所有对象的渲染都是通过调用 Render(Object)函数，方便之后对于场景物体的重构。此外，通过使用 PolygonOffset()方法，我将球体的轮廓线绘制完成，详细的方法后文回阐述。

Camera 类和 Lighter 类

```
class Camera {
    static float viewRotation;
public:
    static void setProject(GLfloat width, GLfloat height);
    static void setModel();
    static void rotate(GLfloat radius);
};

class Lighter {
    static bool lightSwitch;
public:
    static void init();
    static void spotLightMove(const Point& pos);
    static void spotLightSwitch();
};
```

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。

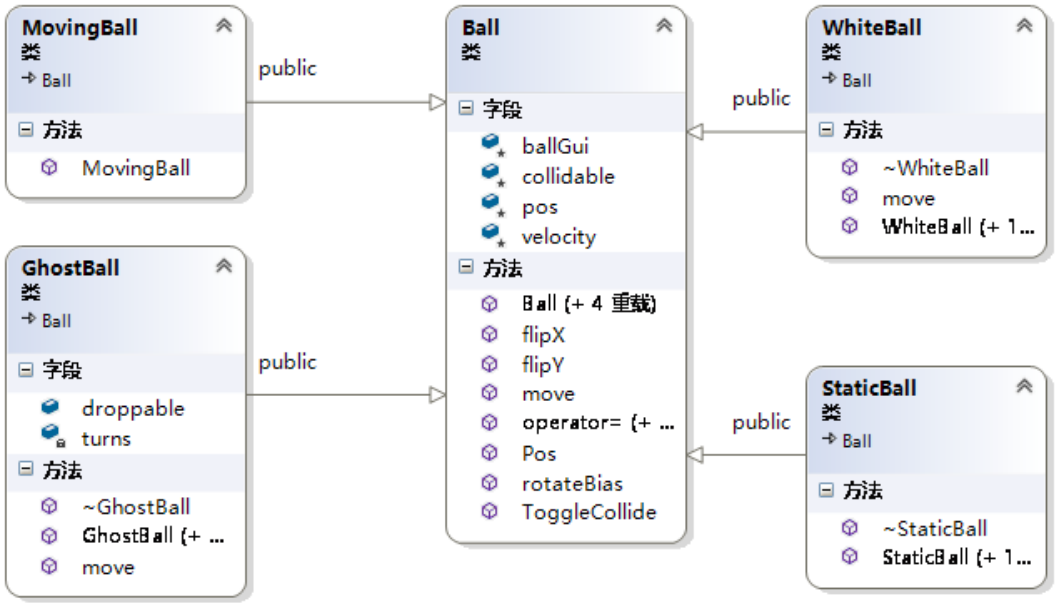
Sound Manager 类

这个类负责播放所有的音效，是一个静态方法类，负责在合适的时候播放对应的音效。目前整个游戏有三个音效，第一个是游戏开始的音效，第二个是得分的音效，第三个是游戏结束的音效。三个音效都取自于超级马里奥的音效。

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、文本、旗帜、天空盒和粒子发射器，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地砖、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。

Tile 类

这个类用来表示的是地砖，除了四个角的坐标之外，还需要记录它的梯度和法向量。

Wall 类

这个类表示天空盒的一面墙，它会记录四个角的坐标和纹理的编号，以便在 render 的时候直接绑定。

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。

Flag 类

这个类用来表示一面旗帜，它包括渲染方法和运动方法。在运动方法中，实现了通过弹簧质点模型进行的布料模拟，详细算法在后文中有详细阐述。

Emitter 类

这个类表示粒子发射器，是用来发射粒子的。在这个类中，需要读入一个方向，然后通过加上一个随机的扰动向量和随机的存活时间来新建一个粒子。并且在运动的时候，如果某个粒子的存活时间为零了，那么就把这个粒子删除。为了使删除粒子的效率提高，并且粒子也不需要随机访问的能力，因此所有的粒子由一个链表构成。

Particle 类

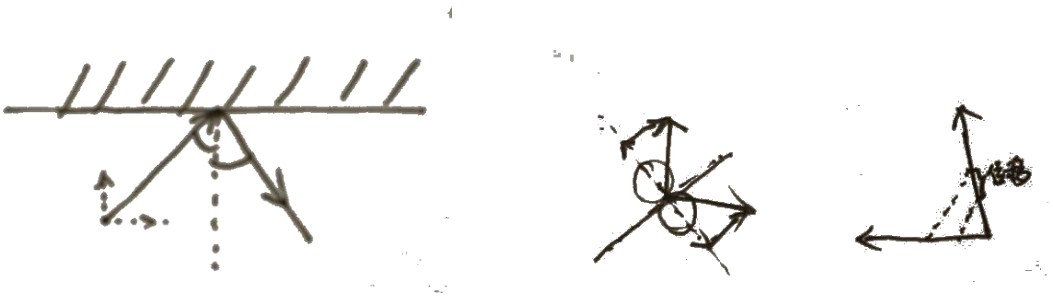
这个类用来表示一个粒子，有自己的位置、速度和存活时间。每次运动会扣除一定的存活时间。

详细算法

物理

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

```
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();
        }
    }
}
```

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。

```
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);
    }
}
```

位置修正

为了实现由于桌面旋转导致对于球的位移（当白球未开球时），特地增加了位置修正函数，从而使得碰撞更为安全，保证桌面不会和球体重叠。

```
void Physics::positionAmend(Ball& ball, const Desk& desk) {
```

```
    for each (auto edge in desk.edges) {
        if (edge.start.x < 0)
            ball.pos.x = ball.pos.x > edge.start.x + ballRadius? ball.pos.x :
edge.start.x + ballRadius;
        else
            ball.pos.x = ball.pos.x < edge.start.x - ballRadius ? ball.pos.x :
edge.start.x - ballRadius;

        if (edge.start.y < 0)
            ball.pos.y = ball.pos.y > edge.start.y + ballRadius ? ball.pos.y :
edge.start.y + ballRadius;
        else
            ball.pos.y = ball.pos.y < edge.start.y - ballRadius ? ball.pos.y :
edge.start.y - ballRadius;
    }
}
```

高度修正

在施加了重力之后，球可能会因为重力的缘故陷入地面，为了防止这个产生的视觉上的错觉，我加入了高度上的修正，将陷入地面的球修正至地面之上。此外，当白球仍未启动的时候，白球应该能够跟着起伏的地表一起上下移动，对于这种特殊情况，我也将白球保持在地面刚好的地方。在算法上，就是先算出球与地面的举例，将其与标准的情况即半径进行相减，减下来的长度是平面法向量的方向，再将其投影至竖直方向，从而完成竖直方向上的移动。

```
bool Physics::heightAmend(Ball& ball, const Desk& desk, bool down) {
    int xIndex = floor(ball.Pos().x) + 5;
    int yIndex = floor(ball.Pos().y) + 5;
    if (xIndex < 0 || xIndex >= 10 || yIndex < 0 || yIndex >= 10)
        return false;
    Tile t = desk.tiles[xIndex][yIndex];
    GLfloat zz = t.leftTop.z + (ball.Pos().x - xIndex + 5) * (t.rightTop.z -
t.leftTop.z) + (ball.Pos().y - yIndex + 5) * (t.leftBottom.z - t.leftTop.z) +
abs((ballRadius - Vec(0, 0, ballRadius) * t.normal) / t.normal.z);
    if (ball.Pos().z < zz) {
        ball.pos.z = zz;
        return true;
    }
    if (ball.Pos().z > zz && !down && !ball.collidable) {
        GLfloat l = abs((Vec(0, 0, ball.Pos().z - zz) * t.normal) /
t.normal.z);
        ball.pos.z = zz;
        return true;
```

```
    }
    return false;
}
```

转动轴计算

在通过对于乒乓球的模拟之后，我发现这是一个非常复杂的模型，特别是球在空中的时候旋转的轴和旋转速度很难界定，因此我简化了整个模型，使得在空中的时候依然和在陆地上的旋转一样，而不考虑特殊情况。在陆地上的时候，转动轴是这样算的：首先获得球的速度向量，其次获得球所在平面的法向量，接着将两者叉乘，叉乘的结果是一个垂直于两者的向量。而这个向量，就是球旋转的旋转轴。（代码中'/'重载为叉乘）

```
void Physics::normalCalc(Ball& ball, const Desk& desk) {
    int xIndex = floor(ball.Pos().x) + 5;
    int yIndex = floor(ball.Pos().y) + 5;
    if (xIndex < 0 || xIndex >= 10 || yIndex < 0 || yIndex >= 10)
        return;
    Tile t = desk.tiles[xIndex][yIndex];
    ball.normal = ball.velocity / t.normal;
}
```

重力施加

在重力施加中，我没有采用普通的先施加向下重力，然后再根据向下移动后可能产生的重叠进行反弹的计算，我认为这个计算复杂且效率底下，因此我进行了一点改进。当球在某个平面上时，计算该平面的梯度，而重力就只沿这个进行分解，从而简化了碰撞检测的过程。

```
void Physics::gravityApply(Ball& ball, const Desk& desk) {
    if (!ball.collidable)
        return;
    int xIndex = floor(ball.Pos().x) + 5;
    int yIndex = floor(ball.Pos().y) + 5;
    if (xIndex < 0 || xIndex >= 10 || yIndex < 0 || yIndex >= 10)
        return;
    Tile t = desk.tiles[xIndex][yIndex];
    GLfloat zz = t.leftTop.z + (ball.Pos().x - xIndex + 5) * (t.rightTop.z -
t.leftTop.z) + (ball.Pos().y - yIndex + 5) * (t.leftBottom.z - t.leftTop.z) +
abs((ballRadius - Vec(0, 0, ballRadius) * t.normal) / t.normal.z);
    if (ball.Pos().z > zz) {
        ball.velocity.z -= ball.gravity;
    }
    if (ball.Pos().z == zz) {
        GLfloat l = Vec(0, 0, -ball.gravity) * t.gradient;
        ball.velocity = ball.velocity + l * t.gradient;
```

```
    }
}
```

（Non-photorealistic rendering，NPR）

对于非真实性渲染，一共分为两个步骤，一是勾边，二是卡通着色。在这个程序中，我着重实现的是勾边的处理，主要采用 PolygonOffset 的方法，并将图形分为两类分类处理。首先第一类是长方体，由于长方体的线框图刚好可以用来当作长方体的勾边，因此直接采用两步式的渲染，先关闭光照渲染线框，再将 PolygonOffset 设为 1 从而渲染内部填充。

```
void Renderer::render(const Desk& desk) {
    glEnable(GL_LIGHTING);
    renderFill(desk);
    glDisable(GL_LIGHTING);
    renderOutline(desk);
}
void Renderer::renderOutline(const Desk& desk) {
    for each (auto rect in desk.rects) {
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        glPushMatrix();
        glTranslatef(rect.x, rect.y, rect.z);
        glScalef(rect.w, rect.l, rect.h);
        glColor3f(0.0, 0.0, 0.0);
        glutWireCube(1.0);
        glPopMatrix();
        glPopAttrib();
    }
}
void Renderer::renderFill(const Desk& desk) {
    for each (auto rect in desk.rects) {
        glPushAttrib(GL_ALL_ATTRIB_BITS);
        glPushMatrix();
        glTranslatef(rect.x, rect.y, rect.z);
        glScalef(rect.w, rect.l, rect.h);
        glEnable(GL_POLYGON_OFFSET_FILL);
        GLfloat ambient[3] = { 0.4, 0.4, 0.4 };
        GLfloat specular[3] = { 1.0, 1.0, 1.0 };
        glMaterialfv(GL_FRONT, GL_AMBIENT, ambient);
        glMaterialfv(GL_FRONT, GL_SPECULAR, specular);
        glPolygonOffset(1.0, 1.0);
        glColor3f(1.0, 1.0, 1.0);
        glutSolidCube(1.0);
        glDisable(GL_POLYGON_OFFSET_FILL);
        glPopMatrix();
        glPopAttrib();
```

```
    }
}
```

另一种是球体、旗帜等线框图无法当作勾边轮廓的情况，这时候采用先将模型扩大向外绘制外框，再缩小绘制实体的方式。这种情况下，抗锯齿设置会使得外框坑洼不理想，因此临时关闭抗锯齿模式。

```
void Renderer::render(const Ball& ball) {
    glPushAttrib(GL_ALL_ATTRIB_BITS);
    glPushMatrix();
    glDisable(GL_LINE_SMOOTH);
    glDisable(GL_BLEND);
    glTranslatef(ball.pos.x, ball.pos.y, ball.pos.z + ballRadius);
    glEnable(GL_POLYGON_OFFSET_FILL);

    // Outline
    glDisable(GL_LIGHTING);
    glPolygonOffset(-3.0f, -3.0f);
    glPolygonMode(GL_FRONT_AND_BACK, GL_LINE);
    glLineWidth(3.0f);
    glColor3f(0.0f, 0.0f, 0.0f);
    glutSolidSphere(ballRadius, 40, 40);

    // Fill
    glEnable(GL_LIGHTING);
    glPolygonMode(GL_FRONT_AND。_BACK, GL_FILL);
    glMaterialfv(GL_FRONT, GL_AMBIENT, ball.ballGui.ambient);
    glMaterialfv(GL_FRONT, GL_SPECULAR, ball.ballGui.specular);
    glColor3f(1.0, 0.0, 0.0);
    glutSolidSphere(ballRadius, 40, 40);

    glDisable(GL_POLYGON_OFFSET_FILL);
    glEnable(GL_LINE_SMOOTH);
    glEnable(GL_BLEND);
    glPopMatrix();
    glPopAttrib();
}
```

旗帜建模

在本程序中，旗帜由 Bezier 曲面构成。对于旗帜，一共有 6x6=36 个控制点对曲面的形态进行控制。相关的代码片段如下：

```
    glEnable(GL_MAP2_VERTEX_3);
    glEnable(GL_MAP2_TEXTURE_COORD_2);
    for (auto i = 0; i < FLAG_HEIGHT; i++) {
```

```
        for (auto j = 0; j < FLAG_WIDTH; j++) {
            flagPoints[i][j] = Vec(-15 - j * 0.75, -5 + j * 0.75, 2.5 - i * 0.75);
        }
    }
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, FLAG_WIDTH, 0, 1, FLAG_WIDTH * 3, FLAG_HEIGHT,
flag.toGLFloat());
    glMapGrid2f(40, 0.0, 1.0, 40, 0.0, 1.0);
    glEvalMesh2(GL_FILL, 0, 40, 0, 40);
```

使用 Bezier 曲面的好处在于，如果不使用它，要使得旗帜柔滑，必须采用大量的控制点来模拟旗帜的形态，这对后面的布料模拟计算造成了很大的计算量，从而降低渲染的效率。为了在下一个迭代中的碰撞检测按时完成，这里优化效率是十分值得的。此外，Bezier 曲面依然提供了非常好的效果，甚至我认为并不输于采用大量控制点模拟的旗帜效果。

旗帜贴图

旗帜的贴图我采用纹理的形式实现，首先这里采用了 SOIL 库来将图片信息导入到材质中，它的使用方式非常简单：

```
int loadImage(void) {
    tex_2d = SOIL_load_OGL_texture("quidditch.png", SOIL_LOAD_AUTO,
SOIL_CREATE_NEW_ID, SOIL_FLAG_INVERT_Y);
    if (tex_2d == 0)
        return -1;

    glBindTexture(GL_TEXTURE_2D, tex_2d);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

    return 0;
}
```

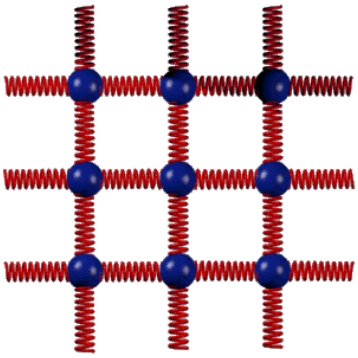
至此，图片相关的材质就被读取到了 tex_2d，且绑定到了 GL_TEXTURE_2D 中。

接着就在 Bezier 曲面绘制时，同时进行材质座标的计算：

```
    glEnable(GL_MAP2_TEXTURE_COORD_2);
    GLfloat texpts[2][2][2] = { { { 0.0, 0.0 },{ 0.0, 1.0 } }, { { 1.0,
0.0 },{ 1.0, 1.0 } } };
    glMap2f(GL_MAP2_TEXTURE_COORD_2, 0, 1, 2, 2, 0, 1, 4, 2, &texpts[0][0][0]);
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, tex_2d);
    glEvalMesh2(GL_FILL, 0, 40, 0, 40);
    glDisable(GL_TEXTURE_2D);
```

由于同时对 Vertex 和 Texture Coord 执行了 glMap2f()操作，这使得之后的 glEvalMesh2()函数能够同时进行两者的计算，从而绑定相应材质。

布料模拟（弹簧质点模型）



对于布料的模拟并不是一个简单的事情，通过对于布料的观察和文献的阅读，我最终采取了一种简单而又有效的模型，将布料想象成理想的弹簧质点集合，从而进行运动。

模型类似于左图，但是与之不同的是，为了仿真的程度更高，我将每个点于周围24个点进行连结,进行扭矩和弹力的计算，从而形成位移。

这个算法的伪代码如下：

```
VECTOR: MovementVector
VECTOR: SpringVector
VECTOR: ForceVector
VECTOR: Gravity
NUMBER: Length
NUMBER: ForceScaler
NUMBER: NormalLength
For every point (p,q) on the cloth:
    MovementVector = Gravity
    For each of the 24 neighbouring points (NB obviously less
at edges)
        SpringVector = (position in space of neighbour) -
(position in space of point (p,q))
        Length       = length of SpringVector
        NormalLength = The length SpringVector would be if the
cloth were unstretched
        ForceScaler  = (Length - NormalLength) / NormalLength
        SpringVector = SpringVector * (1/Length)
        ForceVector  = SpringVector * ForceScaler
        ForceVector  = ForceVector * SmallAmount
        add ForceVector to MovementVector
    end of loop
    Add MovementVector to cloth1(p,q) and store it in
cloth2(p,q)
    make sure this point does not move inside an object
end of loop

Copy all the values in cloth2 to cloth1
```

最终在程序中的实现如下：

```
void Flag::move() {
    GLfloat cgravity = 0.075;
    GLfloat c = 0.125;
    std::vector<std::vector<Vec>> points2(FLAG_HEIGHT,
```

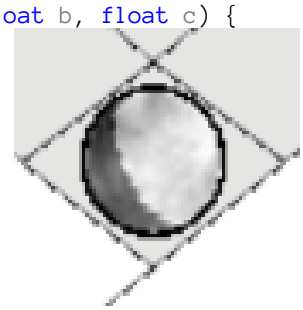
```
std::vector<Vec>(FLAG_WIDTH));
    GLfloat normalPoint[4] = { 1.0, 1.414, 2.336, 2.828 };
    for (auto i = 0; i < FLAG_HEIGHT; i++) {
        for (auto j = 0; j < FLAG_WIDTH; j++) {
            Vec movement = Vec(0, 0, -cgravity);
            for (auto di = -2; di < 3; di++) {
                for (auto dj = -2; dj < 3; dj++) {
                    if (di == 0 && dj == 0)
                        continue;
                    auto ii = i + di, jj = j + dj;
                    if (ii < 0 || ii > FLAG_HEIGHT - 1 || jj < 0 || jj >
FLAG_WIDTH - 1)
                        continue;
                    auto spring = flagPoints[ii][jj] - flagPoints[i][j];
                    auto length = spring.length();
                    auto normal = normalPoint[abs(di) + abs(dj) - 1] * 0.75;
                    auto scaler = (length - normal) / normal;
                    spring = length == 0 ? Vec() : (spring / length);
                    auto force = scaler * c * spring ;
                    movement = force + movement;
                }
            }
            points2[i][j] = flagPoints[i][j] + movement;
        }
    }
    for (auto i = 0; i < FLAG_HEIGHT; i++) {
        for (auto j = 0; j < FLAG_WIDTH; j++) {
            if (i == 0 && j == 0 || i == 0 && j == FLAG_WIDTH - 1)
                continue;
            flagPoints[i][j] = points2[i][j];
        }
    }
}
```

Perlin 噪声函数的应用

随机数会使得程序充满不可知性，但是随机也会使得程序变得不自然，那么如何使程序变得自然呢，就需要采用Perlin噪声函数进行计算。在通过Perlin噪声函数读入一个二维坐标，经过多个维度的叠加之后能够返回一个较为自然的值，我将其作为灰度值，这样就会产生一个类似于大理石纹理的效果，如下：

Perlin噪声函数结果

在获得这个二维纹理之后，再通过 OpenGL 自动运算绑定到球面上，即可产生有大理石纹理的球了：



```
float InterLinear(float a, float b, float c) {
    return a*(1 - c) + b*c;
}

float Noise(int x) {
    x = (x << 13) ^ x;
    return (((x * (x * x * 15731 + 789221) + 1376312589) & 0x7fffffff) /
2147483648.0);
}

float PerlinNoise(float x, float y, int width, int octaves, int seed, double
period) {

    double a, b, value, freq, zone_x, zone_y;
```

```
    int s, box, num, step_x, step_y;
    float amplitude = period;
    int noisedata;

    freq = 1 / period;
    value = 0;

    for (s = 0; s<octaves; s++) {
        num = static_cast<int>(width*freq);
        step_x = static_cast<int>(x*freq);
        step_y = static_cast<int>(y*freq);
        zone_x = x*freq - step_x;
        zone_y = y*freq - step_y;
        box = step_x + step_y*num;
        noisedata = (box + seed);
        a = InterLinear(Noise(noisedata), Noise(noisedata + 1), zone_x);
        b = InterLinear(Noise(noisedata + num), Noise(noisedata + 1 + num),
zone_x);
        value += InterLinear(a, b, zone_y)*amplitude;
        freq *= 4;
        amplitude /= 4;
    }
    return value;
}
```

动态模糊

动态模糊可以给画面带来动感，而其实现主要采用的是帧累加的方式。首先需要减弱当前的累加帧，接着把当前的帧缓存加入，然后绘制整个图像。

```
    auto q = 0.75;
    glAccum(GL_MULT, q); // 将当前累积缓存做不透明度75%处理
    glAccum(GL_ACCUM, 1 - q); // 将当前颜色缓存做不透明度25%处理，并累加
    glAccum(GL_RETURN, 1.0); // 将累积缓存里面的数据发送给颜色缓存，以显示效果
```