台球动画报告

一、作业简介

这是一个模拟台球的程序。通过键盘←→键控制球杆的角度，按下键盘↑↓控制球杆距球的距离，距离越远，击球的力度越大。决定好后按下鼠标左键即可击球，鼠标右键可以切换视角。

二、代码说明

1.程序框架

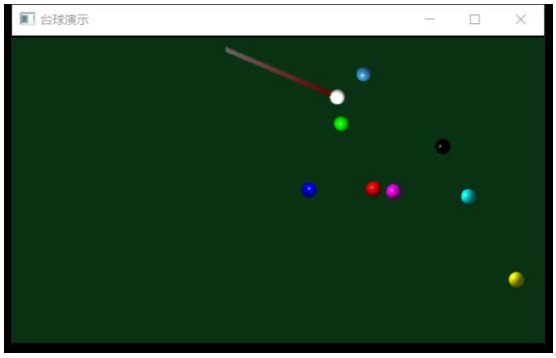
程序的基本函数有 OnDisplay、OnIdle、OnKey、OnMouse、OnReShape 和 SetupLights。其中，OnDisplay 的作用是绘制一幅静止的画面，而 OnIdle 的作用是改变物体的位置，当通过两个函数的不断循环，物体的位置会不断改变，而无数幅静止画面的快速切换即形成动画。OnKey 和 OnMouse 中分别编有当特定键盘按键或鼠标按键被按下后被触发的事件，这里通过这两个函数实现人机的交互。OnReShape 是当窗口的大小被改变后被触发的函数，这里用来保持画面比例的一致。SetupLights 是光照设定。

2.属性的说明

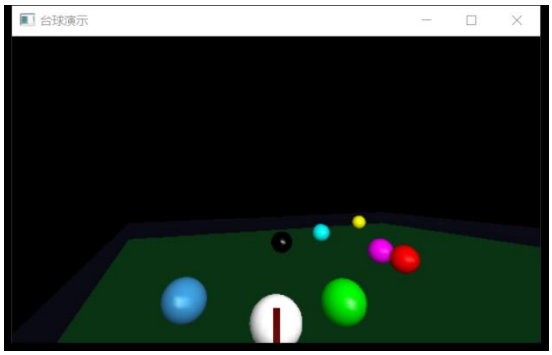
球体具有五个属性：质量、半径、速度（横纵分量）、位置（横纵坐标）、颜色（RGB）。其中，速度的意义为每一帧球体坐标的改变量，同理加速度值的意义为每一帧球体速度的改变量。

3.视角的设定与变换

动画两个视角的切换通过改变全局变量 viewType 的值来实现。



viewType=0 时的正交投影



viewType=1 时的透视投影

当按下鼠标右键的时候，动画的视角会在二者中间相互切换。OnMouse 函数中相关代码如下：

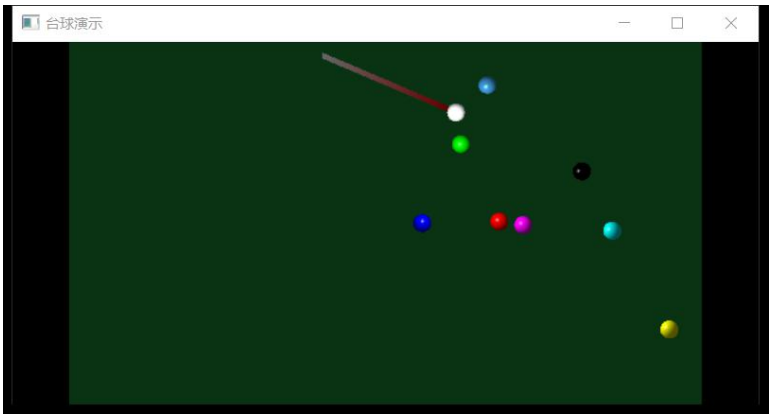
```
    case GLUT_RIGHT_BUTTON://按键为右键
        if (state == GLUT_DOWN){//按下
            if (viewType == 0) {
                viewType = 1;
                OnReShape(windowsWidth, windowsHeight);
            }
            else if (viewType == 1) {
                viewType = 0;
                OnReShape(windowsWidth, windowsHeight);
            }
        }
        break;
```

视角在 OnReShape 函数中设定。OnReShape 函数中代码如下：

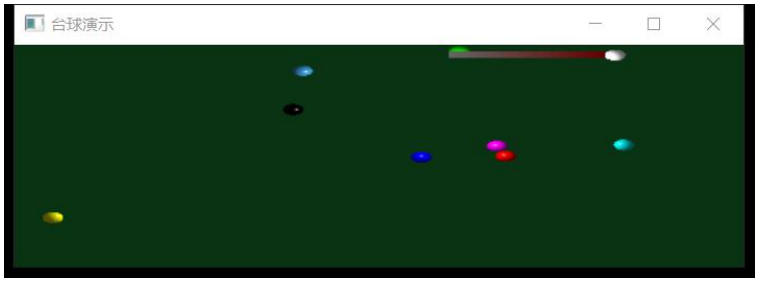
```
void OnReShape(int w, int h){
    GLfloat aspect = (float)w / (float)h;//计算窗口宽高比
    glViewport(0, 0, w, h);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (0 == h) h = 1;
    if (viewType == 0){//正投影
        if (aspect >= SIZE_X / SIZE_Y)    glOrtho(-SIZE_Y*aspect, SIZE_Y*aspect, -SIZE_Y,
SIZE_Y, 0.0f, 250.0f);
        else    glOrtho(-SIZE_X, SIZE_X, -SIZE_X / aspect, SIZE_X / aspect, 0.0f, 250.0f);
    }
    else if (viewType == 1){//透视投影
        gluPerspective(60.0f, aspect, 1.0f, 1000.0f);
    }
    windowsWidth = w;
    windowsHeight = h;
}
```

为了使得在改变窗口宽高比例的时候，实际画面内容的比例不发生改变，我们需要在窗口改变的同时对投影范围进行适应性调整，因此需要用 aspect 存储当前窗口的宽高比。

在透视投影视角中，只需将 aspect 作为 gluPerspective 的参数即可。而在正交投影视角中，我们需要将桌面的全部内容呈现出来，因此这里需要判断：当窗口宽度过大（窗口宽高比>桌面横纵比）时，要增大水平方向的投影范围；当窗口高度过大（窗口宽高比<桌面横纵比）时，要增大竖直方向的投影范围，如图：



进行投影范围调整的效果



未进行投影范围调整的效果

然而在 glutReshapeFunc 中，当且仅当窗口的尺寸发生改变，OnReShape 才会被触发，因此仅仅单击鼠标右键而不改变窗口尺寸，OnReShape 不会被调用，投影类型也就不会被刷新。因此在 OnReShape 的最后，我们将窗口的宽度（windowsWidth）和高度（windowsHeight）储存下来，在 OnMouse 函数中调用 OnReShape 函数将其传入，这样就刷新了投影类型，只单击右键就可以切换视角。

4.击球阶段动画实现

由于在实际台球比赛中，不应在球运动的时候进行击球，因此对球杆的操作这里用全局变量 controllable 进行限制，只有当 controllable==1（击球阶段）的时候，键盘按键和鼠标左键才会被相应。在击球阶段方向键每被按下一次，球杆距球距离（cueDistance）或球杆角度（cueAngle）便会被减小或增大一个固定的量。

当决定好球杆距离和角度后，单击鼠标左键即可击球，相关代码如下：

```
case GLUT_LEFT_BUTTON:
    if (state == GLUT_DOWN && controllable){
        controllable = 0;
        hitDistance = cueDistance;//记录球杆击打距离
        MAIN_BALL.vx = -power*cosf(radian);//赋予主球速度属性
        MAIN_BALL.vy = -power*sinf(radian);
        cueIsMoving = 1;
    }
    break;
```

鼠标左键按下即视为击球阶段的结束，不可再控制球杆，因此要将 controllable 赋值为 0.

在实际击球时，力量越大，球杆运动的速度就越大，因此这里假定每次击球过程中所需

时间相同，这样球杆距离越远，球杆运动就越快；距离越近，运动就越慢。因为球杆每帧改变量和击球（按下鼠标左键）时球杆距离（cueDistance）有关，而 cueDistance 会随着球杆运动而不断减小，因此这里用全局变量 hitDistance 进行击球距离的记录。cueMove 函数中改变 cueDistance 的语句为：

```
cueDistance -= (hitDistance -MAIN_BALL.radius) / CUE_MOVE_FRAME;
```

其中 MAIN_BALL.radius 为主球的半径，CUE_MOVE_FRAME 为球杆运动的帧数常量。

在利用 distanceToPower 函数将球杆距离（cueDistance）转化为击球速度后，将计算后水平和竖直分量上的速度分量赋值到主球的属性中，并将球杆的运动状态（cueIsMoving）赋值为 1。当球杆距离（cueDistance）==主球半径（MAIN_BALL.radius）的时候，球杆击打到了主球，球杆的运动状态（cueIsMoving）被赋值为 0，此时开始滚球的运动。

5.滚球阶段动画实现

OnIdle 函数中，有关球体运动的代码如下：

```
        for (i = 0; i < BALL_QUANTITY; i++){
            accelarateOfRub(&balls[i]);//桌面摩擦力
            if (balls[i].vx != 0 || balls[i].vy != 0)    isStopped = 0;
            moveBall(&balls[i]);//坐标变化
            for (j = i - 1; j >= 0; j--){
                ballCrash(&balls[i], &balls[j]);//球体碰撞判定
            }
            sideCrash(&balls[i]);//边界碰撞判定

        }
        if (isStopped){
            controllable = 1;
        }
```

其中，BALL_QUANTITY 为球体数目常量（包括主球），这里用 for 循环遍历所有的球体。

首先 accelarateOfRub 函数模拟了桌面滑动摩擦力对球体速度的影响，它根据球体速度方向计算出水平和竖直方向的加速度分量，再将速度和加速度相减得到减速后球体的速度。

随后，球体需进行碰撞判定——球体碰撞和边界碰撞。

球体碰撞判定函数 ballCrash 代码如下：

```
void ballCrash(struct ball *p, struct ball *q){
    float vertical1, parallel1, vertical2, parallel2;
    float vertical1_crash, vertical2_crash;
    float angleSin, angleCos;
    float ballDistance;
    ballDistance = distance(p->px, p->py, q->px, q->py);
    if (ballDistance <= p->radius + q->radius){
        angleCos = (p->px - q->px) / ballDistance;//计算两球切线角度
        angleSin = (p->py - q->py) / ballDistance;

        vertical1 = p->vx*angleCos + p->vy*angleSin;//计算垂直切线速度
        vertical2 = q->vx*angleCos + q->vy*angleSin;
```

```
            parallel1 = p->vx*(-angleSin) + p->vy*angleCos;//计算平行切线速度
            parallel2 = q->vx*(-angleSin) + q->vy*angleCos;

            vertical1_crash = ((p->mass - q->mass)*vertical1 + 2 * q->mass * vertical2) / (p->mass
+ q->mass);//完全弹性碰撞，得到碰撞后垂直切线速度
            vertical2_crash = ((q->mass - p->mass)*vertical2 + 2 * p->mass * vertical1) / (p->mass
+ q->mass);

            p->vx = vertical1_crash*angleCos - parallel1*angleSin;//求出碰撞后两球速度属性
            p->vy = vertical1_crash*angleSin + parallel1*angleCos;
            q->vx = vertical2_crash*angleCos - parallel2*angleSin;
            q->vy = vertical2_crash*angleSin + parallel2*angleCos;

            while (distance(p->px, p->py, q->px, q->py) <= p->radius + q->radius){//避免粘连
                moveBall(p);
                moveBall(q);
            }
        }
}
```

当两球距离≤两球半径之和的时候，即视为发生了一次碰撞，此时根据两球坐标，计算出两球在桌面上切线的角度，并根据切线的角度，计算出两球垂直于切线的速度分量大小，与平行于切线的速度分量大小。其中，平行分量不发生变化，而垂直分量根据“完全弹性碰撞计算公式”：

$$v_1' = \frac{(m_1 - m_2)v_1 + 2m_2v_2}{m_1 + m_2}$$

$$v_2' = \frac{(m_2 - m_1)v_2 + 2m_1v_1}{m_1 + m_2}$$

计算出碰撞之后速度的垂直分量大小，再根据平行分量反求回横纵速度分量，将其赋值到两球的属性上，这样就求出了碰撞之后两球的速度。

最后，因为判定碰撞的条件包含两球距离<两球半径之和的可能性，因此我们需要调用 moveBall 函数来使两球运动，直到两球分开，否则可能导致两球一直保持碰撞状态。

对于边界碰撞的判定则较为简单，当发生碰撞时，只需将球体的横向或纵向速度变更为其相反数即可。

当检测到所有球体的速度都为 0 的时候，该回合结束，可以继续击球，此时将 controllable 赋值为 1，如此循环。