数据结构与算法分析 project 项目报告书

小组成员：11912015 高政 11911625 马晨昊 1191628 未韶辉

2021 年 12 月 12 日

1 项目简介

1687 年，艾萨克·牛顿爵士在他著名的《原理》中制定了控制两个粒子在相互引力作用下运动的原理。然而，牛顿无法解决三个粒子的问题。事实上，一般来说，三个或更多粒子的系统只能通过数值求解。本次 project 的目标是编写一个程序来模拟平面中 N 个粒子的运动，受重力相互影响，并对结果进行动画处理。

NBody 系统的基本特性如下：

- 所有对象都是刚体。当它们相互碰撞时，它们不会改变形状。他们也不会碎成碎片。
- 每个物体都有自己的大小、质量、位置和颜色。物体的大小、质量和颜色不会改变。但是位置可以随时间变化。
- 由于物体的位置可以改变，它们可以相互碰撞并相互弹开。当这种情况发生时，能量和动量守恒。
- 物体之间存在重力。这会影响粒子的移动。

我们添加以下简化：

- 在空间中运动的所有物体都是密度均匀的球体，这意味着球的质心是同一个球的几何中心。
- 忽略相对论效应、量子效应、空气阻力（因为空间是真空）、摩擦。
- 两个物体之间的引力表示为：$F = G\frac{M_1 M_2}{R^2}$，其中 G 是引力常数，约为 $6.67\times10^{-11}Nm^2/kg^2$。$M_1$ 和 M_2 分别是两个物体的质量。R 是两者的质心之间的距离。
- 物体光滑。并且它们不会在模拟中旋转。

- 对象位于二维空间中。空间周围有四堵墙。墙壁也是刚体。当物体与墙壁碰撞时，它们可以从墙壁上弹开。墙壁的质量非常小，因此可以忽略墙壁和物体之间的重力。墙壁不会移动，因此仅模拟有限大小的正方形空间，而不是从负无穷大到正无穷大。
- 对象不能相互重叠。物体不能与墙相交。
- 所有碰撞都会立即发生。

2 项目思路

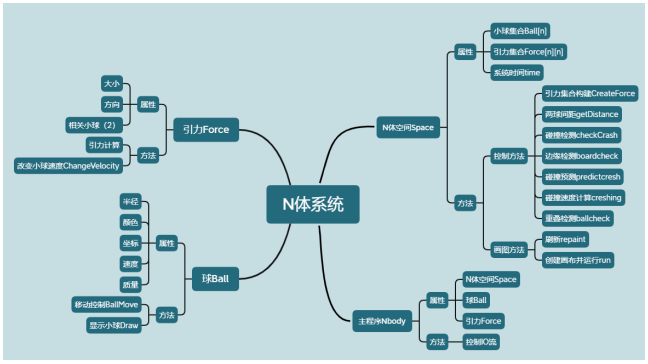


图 1: NBody 仿真 project 项目结构图

2.1 基本元素构建

Ball 定义基本属性：半径，颜色，位置，速度质量。随后构建方法控制小球移动，考虑到速度变化，采用极短时间 t 的位置变化，即 $X = x + t * v$。调用 StdDraw 方法画出该小球对象。

Force 定义基本属性：两个小球来构建之间的引力，引力大小，方向——即 X，Y 方向的正负，引力常量 G。构建方法，根据公式计算两小球之间的引力大小，根据坐标变换确定小球的相对位置，从而确定引力方向。构建控制速度变化的方法，考虑到加速度随时间变化，计算极端时间内，加速度不变时速度的变化，即 $v = v + t * a$。

Space 定义基本属性：小球集合，引力集合，画布长度方法构建中，首先定义引力集合的方法，循环遍历小球集合，两两配对构建引力并储存在引力集合的二维数组中。然后定义边缘检测以及边缘碰撞速度变化。其次是碰撞检测，再调用碰撞检测实现碰撞速度变化控制。最后定义方法刷新画布，在极端时间内进行上述操作，更新所有小球的位置画出小球。

2.2 关键问题及方法

2.2.1 边缘检测及速度变缓

思路 首先是对小球进行边缘检测，检测小球是否已经接触到了边缘，在这个问题上，我们通过分情况分别判断小球的横纵坐标在加减小球半径后与边缘的大小关系，进而判断小球是否已经与边缘进行碰撞。在解决边缘碰撞问题时，考虑到边缘不会受到碰撞影响而且小球的运动满足完全弹性碰撞，小球的碰撞可以分为 x 方向的速度和 y 方向的速度进行分类讨论得到解决。

问题解决 首先对边缘进行分类，如果是左右边缘，那么小球在与边缘碰撞时 y 方向速度不变，x 方向速度反向；如果是上下边缘，那么小球在与边缘碰撞时 x 方向速度不变，y 方向的速度反向。这样就可以很好的解决边缘检测和边缘碰撞问题。

关键代码

```
 public boolean checkboardx(Ball ball) {
   return ball.getBallRX() <= ball.getBALL_R() || ball.getBallRX() >=
       1 - ball.getBall_R();
}

public boolean checkboardy(Ball ball) {
   return ball.getBallRY() <= ball.getBALL_R() || ball.getBallRY() >=
       1 - ball.getBall_R();
}

/**
*墙壁的碰撞检测与速度更新。
*/
public void boardcheck(Ball ball) {
   if (checkboardx(ball)) ball.setVelocityX(-ball.getVelocityX());
```

```
    if (checkboardy(ball)) ball.setVelocityY(-ball.getVelocityY());
  }
```

2.2.2 换图展示

思路 调用 StdDraw 先创建空白画布，在小球对象中定义了画出小球的方法，用循环遍历所有小球，将其推送至缓冲区，最后从缓冲去显示在画布上。然后在根据加速度改变速度，根据速度改变小球位置。更新画布。

问题解决 测试中出现速度过快，看不清楚，决定方法结尾加入暂停方法，来时画面更加平滑。为避免刷新率太低导致画面卡顿，经过多次测试，找到最为合适的刷新率 1ms。

关键代码

```
  /**
  * 检查当前时间是否与用户提问的时间相同，误差范围为计算周期
  * 命中时间则在命令行输出
  */
  public void checktime(){
    for (double[] doubles : questionslist) {
      if (time - 0.5 * rate / 1000 < doubles[0] && time + 0.5 * rate /
          1000 > doubles[0]) {
        int index = (int) doubles[1];
        if (!show) {
          //StdOut.println(time);//打印当前时间
          StdOut.println(balls[index].getBallRX() * boardlength + "
              " +
          balls[index].getBallRY() * boardlength + " " +
          balls[index].getVelocityX() * boardlength + " " +
          balls[index].getVelocityY() * boardlength);
          hasanswered++;
        }
      }
    }
  }

  /**
  *计算单元并根据用户输入显示GUI或命令行
```

```
*/
public void repaint(Ball[] balls){
  colculate++;
  int n =balls.length;

  //根据引力更新小球速度
  for (int i =0; i <n; i++){
     for (int j = i+1; j < n; j++){
        forces[i][j].ChangeVelocityX(boardlength);
        forces[i][j].ChangeVelocityY(boardlength);
     }
  }

  //时间检测
  checktime();
  if (show){
     //计算GUI更新的时间，每1ms一更新
     if (colculate%1 == 0){
        StdDraw.clear();
        for (Ball ball : balls) {
           ball.draw();
        }
        //显示缓存 在界面上
            将所有缓存上用bufferGraphics画完的图形只用一次用之前界面上的画笔g展现处理啊
        StdDraw.show();//0,65为图形左上角坐标 65为了不遮挡鼠标
     }
  }

  //更新小球坐标及碰撞检测
  for (Ball ball : balls) {
     boardcheck(ball);
     ballcheck(balls);
     ball.ballMove();
  }

  //更新引力
  for (int i =0; i <n; i++){
     for (int j = i+1; j < n; j++){
        forces[i][j].setForce_x();
```

```
            forces[i][j].setForce_y();
        }
    }

    //每过1ms利用缓存将数组中全部的小球移动+画出+清屏
    try{
        //Thread.sleep(rate);
        time += rate/1000.0;//时间更新
    }catch(Exception e) {
        e.printStackTrace();
    }
}

public void run() {
    if (show){
        StdDraw.setCanvasSize(800, 800);
        StdDraw.enableDoubleBuffering();//GUI缓存区
    }
    while((show ? true:hasanswered < questionslist.length)) {
        repaint(balls);
    }
    System.exit(0);
}
```

2.2.3 两球碰撞检测及速度变化

思路 通过判断两球球心的距离与两球半径和的大小关系判断两球是否碰撞，在检测到两球碰撞时进行碰撞分析。因为小球碰撞满足完全弹性碰撞，在代码部分可以代入速度的完全弹性碰撞公式。

问题解决 考虑到小球不会只有对心正碰的简单情况，还会有两球斜碰的情况发生，所以碰撞问题的解决我们以两球球心连线为 x 轴建立新坐标系 s，对两球在坐标系 s 中进行碰撞分析会容易很多，两球在新坐标系的 x 轴上分速度满足完全弹性碰撞公式，可以直接带入公式。在新坐标系的 y 轴上分速度不变。然后再将速度合成分解到原坐标系中，从而解决问题。

关键代码

```
/**
* 获得两校求间距的平方
* @return 球心间距的平方
*/
public double getDistance(Ball ball1 , Ball ball2){
   return Math.pow(ball2.getBallRX() - ball1.getBallRX(), 2) +
   Math.pow(ball2.getBallRY() - ball1.getBallRY(), 2);
}

/**
* 判断两小球是否相交
*/
public boolean checkCrash(Ball ball1 , Ball ball2) {
   double distance = getDistance(ball1, ball2);
   return distance <= Math.pow(ball1.getBALL_R() + ball2.getBall_R(),
       2);
}

/**
*碰撞预测，只有慢速球在前的碰撞有效。
* @param ball1 待测小球1
* @param ball2 待测小球2
* @return true 碰撞合法
*/
public boolean predictcresh(Ball ball1, Ball ball2){
   double dx = ball1.getBallRX() - ball2.getBallRX();
   double dvx = ball1.getVelocityX() - ball2.getVelocityX();
   double dy = ball1.getBallRY() - ball2.getBallRY();
   double dvy = ball1.getVelocityY() - ball2.getVelocityY();
   return dx*dvx < 0 || dy*dvy < 0;
}

/**
* 针对碰撞过程中小球的坐标和速度的更新
* 坐标变化：小球相互嵌入时令其回退至两球刚好相切
* 速度更新：将小球速度坐标系变为两小球球心连线为横坐标。
*/
public void collide(Ball ball1,Ball ball2){
```

```
//第一步：先判断两个球的位置
if (sqrt(getDistance(ball1,ball2))==0)
return;
if (sqrt(getDistance(ball1,ball2)) > (ball1.getBALL_R() +
    ball2.getBall_R()))
return;

//第二步：计算合速度的大小和方向
//计算ball1的合速度的值（不包含方向）
double ball1v = sqrt((ball1.getVelocityX()*ball1.getVelocityX()) +
    (ball1.getVelocityY()*ball1.getVelocityY()));
//计算ball1的运行角度（运行方向，【-pi~pi】）
double ball1_angle = acos(ball1.getVelocityX()/ball1v*1);
if (ball1.getVelocityY() != 0)
ball1_angle *= ball1.getVelocityY()/abs(ball1.getVelocityY());

//计算ball2的合速度的值
double ball2v =
    sqrt(ball2.getVelocityX()*ball2.getVelocityX()+ball2.getVelocityY()*ball2.getVelocityY());
//计算ball2的运行角度
double ball2_angle = acos(ball2.getVelocityX()/ball2v*1);
if (ball2.getVelocityY() != 0)
ball2_angle *= ball2.getVelocityY()/abs(ball2.getVelocityY());

//第三步：计算共同坐标系s，以及ball1和ball2在s中的速度的大小和方向
//第三步：第一部分：计算共同坐标系s的参数
//计算ball1球心相对于ball2球心的坐标（注意y轴为数学上的y轴，与计算机中的y轴相反）
double sx = ball2.getBallRX()-ball1.getBallRX();
double sy = ball2.getBallRY()-ball1.getBallRY();
//计算ball2相对于ball1的速度方向和大小
double sv = sqrt(pow(sx,2)+pow(sy,2));
double s_angle = acos(sx/sv);
if (sy !=0)
s_angle *=sy/abs(sy);

//第三步：第二部分：解决粘连问题，或者检测到碰撞时已经相交，让两个球分离
//此时ball1的半径加ball2的半径等于两球心的距离时算相切
//两球相交（或两球粘连）
```

```
if (sqrt(getDistance(ball1, ball2))<(ball1.getBALL_R() +
    ball2.getBall_R())){
  double de = ((ball1.getBALL_R() +
      ball2.getBall_R())-sqrt(getDistance(ball1, ball2)))/2;
  double dxx = de * cos(s_angle);
  double dyy = de * sin(s_angle);
  ball1.setBallRX(ball1.getBallRX() - dxx);
  ball1.setBallRY(ball1.getBallRY() - dyy);
  ball2.setBallRX(ball2.getBallRX() + dxx);
  ball2.setBallRY(ball2.getBallRY() + dyy);
}

//第三步：第三部分：将ball1,ball2的速度大小和方向由原坐标系转化为在共同坐标系s中的速度大小和方向
// 在s坐标系中ball1的新运动方向
double s_ball1_angle = ball1_angle - s_angle;
// 在s坐标系中ball1的新速度大小
double s_ball1_v = ball1v;
// 在s坐标系中ball1的新速度在s的x轴上的投影
double s_ball1_vx = s_ball1_v * cos(s_ball1_angle);
// 在s坐标系中ball1的新速度在s的y轴上的投影
double s_ball1_vy = s_ball1_v * sin(s_ball1_angle);

// 在s坐标系中ball2的新运动方向
double s_ball2_angle = ball2_angle - s_angle;
// 在s坐标系中ball2的新速度大小
double s_ball2_v = ball2v;
// 在s坐标系中ball2的新速度在s的x轴上的投影
double s_ball2_vx = s_ball2_v * cos(s_ball2_angle);
// 在s坐标系中ball2的新速度在s的y轴上的投影
double s_ball2_vy = s_ball2_v * sin(s_ball2_angle);

//第四步：发生完全弹性斜碰时，在s坐标系中，两球y轴速度不变，x轴速度满足完全弹性正碰（由动能定理和动量
// 碰撞后ball1的s坐标系x轴的分速度
double s_ball1_vxfinal = ((ball1.getMass() - ball2.getMass()) *
    s_ball1_vx + 2 * ball2.getMass() * s_ball2_vx) /
    (ball1.getMass()+ball2.getMass());
// 碰撞后ball1的s坐标系y轴的分速度
double s_ball1_vyfinal = s_ball1_vy;
// 碰撞后ball2的s坐标系x轴的分速度
```

```
double s_ball2_vxfinal = ((ball2.getMass()-ball1.getMass()) *
    s_ball2_vx + 2 * ball1.getMass() * s_ball1_vx) /
    (ball1.getMass()+ball2.getMass());
// 碰撞后ball2的s坐标系y轴的分速度
double s_ball2_vyfinal = s_ball2_vy;

//第五步：计算两球发生碰撞后在s中的各自合速度大小和运动方向
// 碰撞后ball1在s坐标系的合速度大小
double s_ball1_vfinal = sqrt(pow(s_ball1_vxfinal, 2) +
    pow(s_ball1_vyfinal, 2));
// 碰撞后ball1在s坐标系的运动方向
double s_balll1_anglefinal = acos(s_ball1_vxfinal /
    s_ball1_vfinal);
if (s_ball1_vxfinal == 0 && s_ball1_vyfinal == 0)
s_balll1_anglefinal = 0;
else if (s_ball1_vyfinal != 0)
s_balll1_anglefinal *= s_ball1_vyfinal / abs(s_ball1_vyfinal);
// 碰撞后ball2在s坐标系的合速度大小
double s_ball2_vfinal = sqrt(pow(s_ball2_vxfinal, 2) +
    pow(s_ball2_vyfinal, 2));
// 碰撞后ball2在s坐标系的运动方向
double s_ball2_anglefinal = acos(s_ball2_vxfinal / s_ball2_vfinal);
if (s_ball2_vxfinal == 0 && s_ball2_vyfinal == 0)
s_ball2_anglefinal = 0;
else if (s_ball2_vyfinal != 0)
s_ball2_anglefinal *= s_ball2_vyfinal / abs(s_ball2_vyfinal);

//第六步：将两球速度转化为原坐标系中的速度
// 碰撞后ball1在原坐标系的合速度大小
double final_ball1_v = s_ball1_vfinal;
// 碰撞后ball1在原坐标系的运动方向
double final_ball1_angle = s_balll1_anglefinal + s_angle;
if (final_ball1_angle > PI)
final_ball1_angle -= 2 * PI;
else if (final_ball1_angle <= -PI)
final_ball1_angle += 2 * PI;
// 碰撞后ball1在原坐标系的合速度大小在x轴上的分量
double final_ball1_vx = final_ball1_v * cos(final_ball1_angle);
// 碰撞后ball1在原坐标系的合速度大小在y轴上的分量
```

```
    double final_ball1_vy = final_ball1_v * sin(final_ball1_angle) ;

    // 碰撞后ball2在原坐标系的合速度大小
    double final_ball2_v = s_ball2_vfinal;
    // 碰撞后ball2在原坐标系的运动方向
    double final_ball2_angle = s_ball2_anglefinal + s_angle;
    if (final_ball2_angle > PI)
    final_ball2_angle -= 2 * PI;
    else if (final_ball2_angle <= -PI)
    final_ball2_angle += 2 * PI;
    // 碰撞后ball2在原坐标系的合速度大小在x轴上的分量
    double final_ball2_vx = final_ball2_v * cos(final_ball2_angle);
    // 碰撞后ball2在原坐标系的合速度大小在y轴上的分量
    double final_ball2_vy = final_ball2_v * sin(final_ball2_angle) ;

    // 第七步：更新
    ball1.setVelocityX(final_ball1_vx);
    ball1.setVelocityY(final_ball1_vy);
    ball2.setVelocityX(final_ball2_vx);
    ball2.setVelocityY(final_ball2_vy);
}

/**
* 小球间的碰撞检测与速度更新。
*/
public void ballcheck(Ball[] balls){
    int[][] hascheck = new int[balls.length][balls.length];
    for (int i = 0;i < balls.length;i++){
        for (int j = 0;j < balls.length;j++){
            if (i == j) continue;
            if (checkCrash(balls[i], balls[j])){
                if (hascheck[Math.min(i, j)][Math.max(i, j)] == 0){
                    if (predictcresh(balls[i], balls[j])){
                        collide(balls[i], balls[j]);
                        hascheck[Math.min(i, j)][Math.max(i, j)] = 1;
                    }
                }
                break;
            }
```

```
    }
  }
}
```

2.2.4 引力构建以及加速度变化

思路 通过已有的小球集合，创建一个引力 Force 的 2 维数组，每个小球两两配对，创建对应引力来提供小球的加速度以及速度控制。然后调用 Force 中的速度控制方法，分别控制相关小球的速度变化。

问题解决 在测试过程中产生小球引力方向相反的问题。经研究发现，在引力构建的时候缺少严格的方向控制，即小球位置不同，速度方向不同导致相对引力方向不同。因此利用坐标变换，以其中一个球 A 为坐标轴，计算出另一个 B 的相对位置，然后分别判断 XY 方向上的引力方向。最终问题解决。

关键代码

```
public Force[][] CreateForce(){
  int n = balls.length;
  Force[][] forces = new Force[n][n];
  for(int i =0 ; i < n-1 ; i++){
    for (int j = i+1; j<n ; j++){
      forces[i][j] = new Force(balls[i] , balls[j]);
    }
  }
  return forces;
}

/**
* 更新小球的x轴向速度
*/
public void ChangeVelocityX(int boardlength){
  double ball1_mess = ball1.getMass();
  double ball2_mess = ball2.getMass();
  double ball1_v = ball1.getVelocityX();
  double ball2_v = ball2.getVelocityX();

  ball1_v -= 100* rate * force_x / (ball1_mess * boardlength);
```

```
    ball2_v += 100* rate * force_x / (ball2_mess * boardlength);

    ball1.setVelocityX(ball1_v);
    ball2.setVelocityX(ball2_v);
  }

  /**
  * 更新小球的y轴向速度
  */
  public void ChangeVelocityY(int boardlength){
    double ball1_mess = ball1.getMass();
    double ball2_mess = ball2.getMass();
    double ball1_v = ball1.getVelocityY();
    double ball2_v = ball2.getVelocityY();

    ball1_v -= rate * force_y / (ball1_mess * boardlength);
    ball2_v += rate * force_y / (ball2_mess * boardlength);

    ball1.setVelocityY(ball1_v);
    ball2.setVelocityY(ball2_v);
  }
```

2.2.5 时间控制以及单位归一化

思路 小区状态更新与力的更新受世界时间控制，因此整个系统需要一个统一的时间来控制整体的进程。另一方面 StdDraw 方法的输入区间为 $0 \sim 1$，而用户输入的包括小球坐标、速度、半径等均为标准单位，为使程序能够正常运行，需对所有数据进行归一化操作。

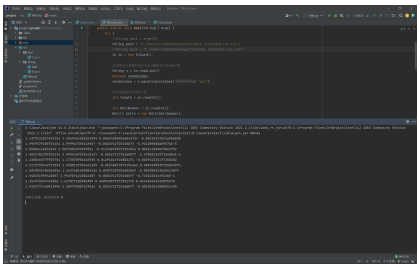
问题解决 通过设置一个全局时间 time，每计算一次全局时间增加一个周期 rate，以此来控制整个系统的时间。同时在创建 Ball 对象时，就通过 $x = x/boaardlength$ 的形式将小球的坐标、半径和速度归一化到 $0 \sim 1$ 之间。

关键代码

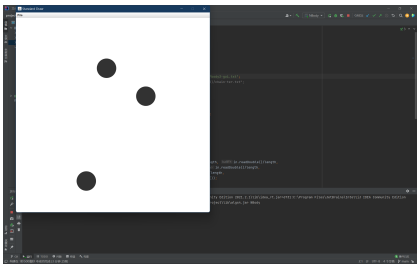
```
  int BallNumber = in.readInt();
  Ball[] balls = new Ball[BallNumber];
```

```
//根据输入生成小球对象时对其参数进行归一化
for (int i = 0;i < BallNumber; i++){
   balls[i] = new Ball(in.readDouble()/length, in.readDouble()/length,
   in.readDouble()/length, in.readDouble()/length,
   in.readDouble(), in.readDouble()/length,
   in.readInt(), in.readInt(), in.readInt());
}
```

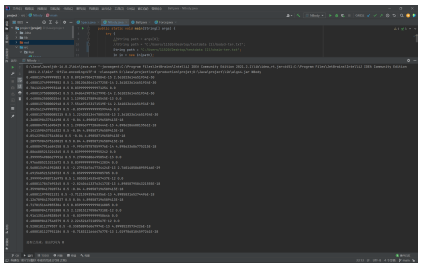
3 项目结果



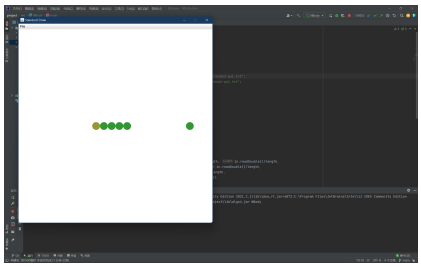
(a) body3_ter



(b) body3_gui



(c) chain_ter



(d) chain_gui

图 2: 测试样例

我们使用了 body3 和 chain 两个测试样例对我们的程序进行测试，测试结果误差控制在 1% 以内，基本符合预期效果。

4 项目总结

在经过几个昼夜的代码编写和 debug 后，我们终于完整的实现了本课程期末项目的所有要求。我们在运行测试样例时遇到过各种各样的 bug，但最后都通过团队合作商讨以及逐步排除的方法解决了问题，对 java 代码在实际问题的解决方面有了更深的经验。同时在过程中我们查阅了许多关于 N 体问题的相关论文和资料，对这方面的知识有了更深的理解。