**Project:** Simulating an elastic collision with 'n' balls

**Group Members:**

Jugat Singh Lamba (14BCE1150)

Ayush Agrawal (14BCE1015)

Swastik Mittal (14BCE1128)

Avnish Garg (14BCE1180)

Rajat Goel (14BCE1168)

**Code:**

[Note: The code is broken into 3 modules: Particle, MinPQ and Collision System]

Particle.java:

import java.awt.Color;

public class Particle{

private static final double INFINITY = Double.POSITIVE\_INFINITY;

private double rx,ry; // position

private double vx,vy; // velocity

private final double radius;

private final double mass;

private int count; // no of collisions

private Color color;

Particle(){

rx = StdRandom.uniform(0.0, 1.0);

ry = StdRandom.uniform(0.0, 1.0);

vx = StdRandom.uniform(-.005, 0.005);

vy = StdRandom.uniform(-.005, 0.005);

radius = 0.01;

mass = 0.5;

color = Color.BLACK;

}

Particle(double a, double b, double c, double d, double r,double m,Color c2){

rx=a;

ry=b;

vx=c;

vy=d;

radius=r;

mass=m;

color=c2;

}

public void move(double dt){

rx+=vx\*dt;

ry+=vy\*dt;

}

public void draw(){

StdDraw.setPenColor(color);

StdDraw.filledCircle(rx,ry,radius);

}

public int count(){

return count;

}

// predict collision

public double timeToHit(Particle that){

if (this == that) return INFINITY;

double dx = that.rx - this.rx;

double dy = that.ry - this.ry;

double dvx = that.vx - this.vx;

double dvy = that.vy - this.vy;

double dvdr = dx\*dvx + dy\*dvy;

if (dvdr > 0) return INFINITY;

double dvdv = dvx\*dvx + dvy\*dvy;

double drdr = dx\*dx + dy\*dy;

double sigma = this.radius + that.radius;

double d = (dvdr\*dvdr) - dvdv \* (drdr - sigma\*sigma);

// if (drdr < sigma\*sigma) StdOut.println("overlapping particles");

if (d < 0) return INFINITY;

return -(dvdr + Math.sqrt(d)) / dvdv;

}

public double timeToHitVerticalWall(){

if (vx > 0) return (1.0 - rx - radius) / vx;

else if (vx < 0) return (radius - rx) / vx;

else return INFINITY;}

public double timeToHitHorizontalWall(){

if (vy > 0) return (1.0 - ry - radius) / vy;

else if (vy < 0) return (radius - ry) / vy;

else return INFINITY;

}

// resolving collision

public void bounceOff(Particle that){

double dx = that.rx - this.rx;

double dy = that.ry - this.ry;

double dvx = that.vx - this.vx;

double dvy = that.vy - this.vy;

double dvdr = dx\*dvx + dy\*dvy; // dv dot dr

double dist = this.radius + that.radius; // distance between particle centers at collison

// normal force F, and in x and y directions

double F = 2 \* this.mass \* that.mass \* dvdr / ((this.mass + that.mass) \* dist);

double fx = F \* dx / dist;

double fy = F \* dy / dist;

// update velocities according to normal force

this.vx += fx / this.mass;

this.vy += fy / this.mass;

that.vx -= fx / that.mass;

that.vy -= fy / that.mass;

// update collision counts

this.count++;

that.count++;

}

public void bounceOffVerticalWall(){

vx = -vx;

count++;

}

public void bounceOffHorizontalWall(){

vy = -vy;

count++;

}

public double kineticEnergy() {

return 0.5 \* mass \* (vx\*vx + vy\*vy);

}

}

MinPQ.java:

import java.util.Comparator;

import java.util.Iterator;

import java.util.NoSuchElementException;

public class MinPQ<Key> implements Iterable<Key> {

private Key[] pq; // store items at indices 1 to N

private int N; // number of items on priority queue

private Comparator<Key> comparator; // optional comparator

public MinPQ(int initCapacity) {

pq = (Key[]) new Object[initCapacity + 1];

N = 0;

}

public MinPQ() {

this(1);

}

public MinPQ(int initCapacity, Comparator<Key> comparator) {

this.comparator = comparator;

pq = (Key[]) new Object[initCapacity + 1];

N = 0;

}

public MinPQ(Comparator<Key> comparator) {

this(1, comparator);

}

public MinPQ(Key[] keys) {

N = keys.length;

pq = (Key[]) new Object[keys.length + 1];

for (int i = 0; i < N; i++)

pq[i+1] = keys[i];

for (int k = N/2; k >= 1; k--)

sink(k);

}

public boolean isEmpty() {

return N == 0;

}

public int size() {

return N;

}

public Key min() {

if (isEmpty()) throw new NoSuchElementException("Priority queue underflow");

return pq[1];

}

private void resize(int capacity) {

Key[] temp = (Key[]) new Object[capacity];

for (int i = 1; i <= N; i++) {

temp[i] = pq[i];

}

pq = temp;

}

public void insert(Key x) {

if (N == pq.length - 1) resize(2 \* pq.length);

pq[++N] = x;

swim(N);

}

public Key delMin() {

if (isEmpty()) throw new NoSuchElementException("Priority queue underflow");

exch(1, N);

Key min = pq[N--];

sink(1);

pq[N+1] = null; // avoid loitering and help with garbage collection

if ((N > 0) && (N == (pq.length - 1) / 4)) resize(pq.length / 2);

return min;

}

private void swim(int k) {

while (k > 1 && greater(k/2, k)) {

exch(k, k/2);

k = k/2;

}

}

private void sink(int k) {

while (2\*k <= N) {

int j = 2\*k;

if (j < N && greater(j, j+1)) j++;

if (!greater(k, j)) break;

exch(k, j);

k = j;

}

}

private boolean greater(int i, int j) {

if (comparator == null) {

return ((Comparable<Key>) pq[i]).compareTo(pq[j]) > 0;

}

else {

return comparator.compare(pq[i], pq[j]) > 0;

}

}

private void exch(int i, int j) {

Key swap = pq[i];

pq[i] = pq[j];

pq[j] = swap;

}

public Iterator<Key> iterator() { return new HeapIterator(); }

private class HeapIterator implements Iterator<Key> {

private MinPQ<Key> copy;

public HeapIterator() {

if (comparator == null) copy = new MinPQ<Key>(size());

else copy = new MinPQ<Key>(size(), comparator);

for (int i = 1; i <= N; i++)

copy.insert(pq[i]);

}

public boolean hasNext() { return !copy.isEmpty(); }

public void remove() { throw new UnsupportedOperationException(); }

public Key next() {

if (!hasNext()) throw new NoSuchElementException();

return copy.delMin();

}

}

public static void main(String[] args) {

MinPQ<String> pq = new MinPQ<String>();

while (!StdIn.isEmpty()) {

String item = StdIn.readString();

if (!item.equals("-")) pq.insert(item);

else if (!pq.isEmpty()) StdOut.print(pq.delMin() + " ");

}

StdOut.println("(" + pq.size() + " left on pq)");

}

}

CollisionSystem.java:

import java.awt.Color;

public class CollisionSystem {

private MinPQ<Event> pq; // the priority queue

private double t = 0.0; // simulation clock time

private double hz = 0.5; // number of redraw events per clock tick

private Particle[] particles; // the array of particles

public CollisionSystem(Particle[] particles) {

this.particles = particles; // .clone() can be used for defensive copying

}

private void predict(Particle a, double limit) {

if (a == null) return;

for (int i = 0; i < particles.length; i++) {

double dt = a.timeToHit(particles[i]);

if (t + dt <= limit)

pq.insert(new Event(t + dt, a, particles[i]));

}

double dtX = a.timeToHitVerticalWall();

double dtY = a.timeToHitHorizontalWall();

if (t + dtX <= limit) pq.insert(new Event(t + dtX, a, null));

if (t + dtY <= limit) pq.insert(new Event(t + dtY, null, a));

}

private void redraw(double limit) {

StdDraw.clear();

for (int i = 0; i < particles.length; i++) {

particles[i].draw();

}

StdDraw.show(10);

if (t < limit) {

pq.insert(new Event(t + 1.0 / hz, null, null));

}

}

public void simulate(double limit) {

pq = new MinPQ<Event>();

for (int i = 0; i < particles.length; i++) { // this is the preprocessing O(n^2) part

predict(particles[i], limit);

}

pq.insert(new Event(0, null, null)); // redraw event

while (!pq.isEmpty()) {

Event e = pq.delMin();

if (!e.isValid()) continue;

Particle a = e.a;

Particle b = e.b;

for (int i = 0; i < particles.length; i++)

particles[i].move(e.time - t);

t = e.time;

if (a != null && b != null) a.bounceOff(b); // particle-particle collision

else if (a != null && b == null) a.bounceOffVerticalWall(); // particle-wall collision

else if (a == null && b != null) b.bounceOffHorizontalWall(); // particle-wall collision

else if (a == null && b == null) redraw(limit); // redraw event

predict(a, limit);

predict(b, limit);

}

}

private static class Event implements Comparable<Event> {

private final double time; // time that event is scheduled to occur

private final Particle a, b; // particles involved in event, possibly null

private final int countA, countB; // collision counts at event creation

public Event(double t, Particle a, Particle b) {

this.time = t;

this.a = a;

this.b = b;

if (a != null) countA = a.count();

else countA = -1;

if (b != null) countB = b.count();

else countB = -1;

}

public int compareTo(Event that) {

if (this.time < that.time) return -1;

else if (this.time > that.time) return +1;

else return 0;

}

public boolean isValid() {

if (a != null && a.count() != countA) return false;

if (b != null && b.count() != countB) return false;

return true;

}

}

public static void main(String[] args) {

StdDraw.setCanvasSize(800, 800);

StdDraw.show(0);

Particle[] particles;

int N = Integer.parseInt(args[0]);

particles = new Particle[N];

for (int i = 0; i < N; i++)

particles[i] = new Particle();

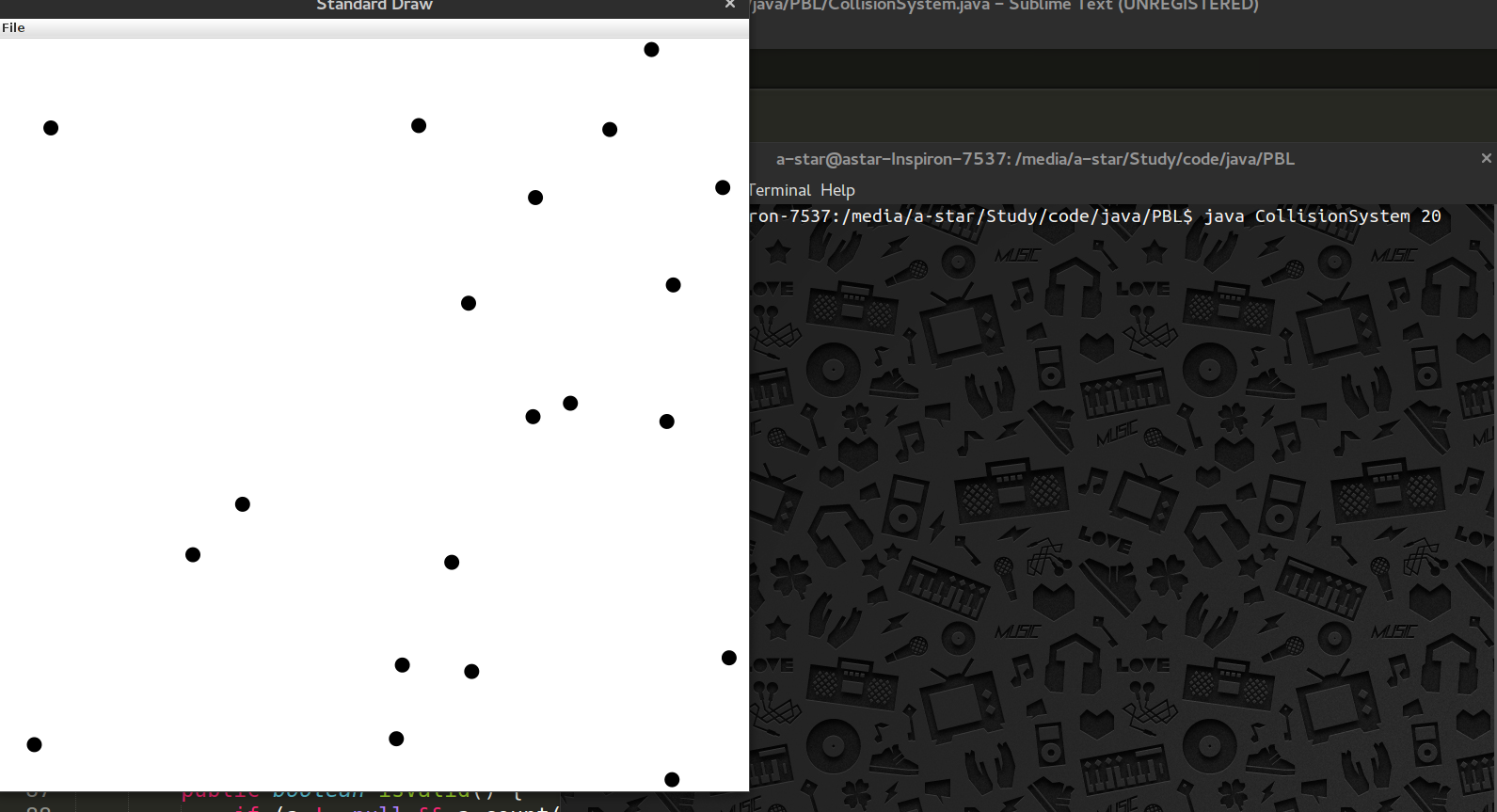
CollisionSystem system = new CollisionSystem(particles);

system.simulate(10000);

}

}

**Screenshot:**

****