École Polytechnique

PSC INF 12

Surveillance de zone par réseau de drones

Annexes

Paul Mortamet
Thibault Vignon
Louis Proffit
Louis Stefanuto
X2019

Tuteur :
M. Vincent Jeauneau

Coordinateur :
M. Emmanuel Haucourt

2020 - 2021

Table des matières

Les codes, ainsi que le reste de notre travail est disponible sur le lien suivant :

https ://github.com/LouisStefanuto/PSC.git

1 Méthode de clustering

Le projet java est divisé en trois packages : celui contenant les algorithmes, celui contenant les classes graphiques (dont *Controller* qui est la classe à exécuter pour faire fonctionner la simulation), et le package structure qui contient les outils pour les autres classes.

1.1 Package algorithms

```
package algorithms;

import java.util.HashMap;
import java.util.LinkedList;

import structure.Checkpoint;
import structure.Cluster;
import structure.Drone;
import structure.Modification;
import structure.Pair;
import structure.Vector;

/**
 * Classe gérant l'attribution des drones aux clusters, grâce
    à un algorithme de
 * Recuit
 *
 * @author Louis Proffit
 * @version 1.0
 */
public class ClusterAttribution implements RecuitInterface {

    /**
     * Liste des drones à attribuer
     */
    private LinkedList<Drone> drones = new LinkedList<>();

    /**
     * Matrice d'association drone <-> cluster. Chaque drone de la
        liste drones est
     * dans la matrice. Toutes les valeurs de la matrice sont
        distinctes.
     */
    private HashMap<Drone, Cluster> association = new HashMap<>();

    /**
     * Objet d'association des checkpoints aux clusters
     */
    private ClusteringSolver clustering = new ClusteringKMeans();
```

```
/**
 * M thode pour ajouter un drone
 *
 * @param drone    : Le drone     ajouter
 * @param improve : Le travail de mise    jour     effectuer
    la fin
 */
public void addDrone(Drone drone, ImproveType improve) {
    drones.add(drone);
    Cluster cluster = new Cluster();
    association.put(drone, cluster);
    clustering.addCluster(cluster, improve);
    TSPRecuit.improvePath(this);
}

/**
 * Renvoie le cluster associ      un drone. Essentiellement
    utilis   par les
 * m thodes graphiques
 *
 * @param drone
 * @return
 */
public Cluster getDroneCluster(Drone drone) {
    return association.get(drone);
}

/**
 * Ajoute un checkpoint
 *
 * @param checkpoint : Le checkpoint     ajouter
 * @param improve    : Le travail    effectuer    la fin
 */
public void addCheckpoint(Checkpoint checkpoint, ImproveType
   improve) {
    clustering.addCheckpoint(checkpoint, improve);
}

/**
 * Am liore la r partition des drones et des clusters. Utilise
    un algorithme de
 * recuit simul .
 *
 * @param improve : Le type d'am lioration     effectuer
 */
public void improve(ImproveType improve) {
    clustering.improve(improve);
    if (improve == ImproveType.COMPLETE)
```

```
            TSPRecuit.improvePath(this);
    }

    /**
     * Renvoie la liste des drones. Essentiellement utilis  par les
        m thodes
     * graphiques
     *
     * @return La liste des drones
     */
    public LinkedList<Drone> getDrones() {
        return drones;
    }

    /**
     * Fait effectuer    tous les drones un mouvement. Si ils
        atteignent leur cible,
     * ils se placent dessus et celle-ci est mise    jour. Si il n'y
        a pas de cible,
     * le drone reste immobile.
     */
    public void move() {
        Vector target;
        Cluster cluster;
        for (Drone drone : drones) {
            cluster = association.get(drone);
            target = cluster.getCurrentTarget();
            if (target == null)
                continue;
            if (target.distance(drone) < Drone.speed) {
                drone.set(target);
                cluster.moveTargetForward();
            } else
                drone.move(target);
        }
    }

    @Override
    public int getSize() {
        return drones.size();
    }

    @Override
    public Modification modificationFunction() {
        return new Pair<Integer>((int) (drones.size() *
            Math.random()), (int) (drones.size() * Math.random()));
    }

    @Override
```

```
    @SuppressWarnings("unchecked")
    public Double improvementFunction(Modification modification) {
        Pair<Integer> swap = (Pair<Integer>) modification;
        Drone firstDrone = drones.get(swap.getFirst());
        Drone secondDrone = drones.get(swap.getSecond());
        double result = 0;
        result += association.get(firstDrone).distance(secondDrone);
        result += association.get(secondDrone).distance(firstDrone);
        result -= association.get(firstDrone).distance(firstDrone);
        result -= association.get(secondDrone).distance(secondDrone);
        return result;
    }

    @Override
    @SuppressWarnings("unchecked")
    public void commitFunction(Modification modification) {
        Pair<Integer> swap = (Pair<Integer>) modification;
        Drone firstDrone = drones.get(swap.getFirst());
        Drone secondDrone = drones.get(swap.getSecond());
        Cluster firstCluster = association.get(firstDrone);
        Cluster secondCluster = association.get(secondDrone);
        association.put(firstDrone, secondCluster);
        association.put(secondDrone, firstCluster);
    }
}
```

```
package algorithms;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.Map.Entry;

import structure.Checkpoint;
import structure.Cluster;

/**
 * La classe est une version de l'algorithme de clustering par
    arbre. Elle met
 * en place un syst me de union find pour identifier les classes
    des clusters Le
 * tableau classes = int[] correspond    cela. Tous les checkpoints
    pointent vers
 * leur parent, qui peut ne pas  tre  un repr sentant de la classe.
    Le tableau
 * distances = double[][] comporte les distances entre clusters, il
    n'est valide
 * que pour tous les repr sentants de checkpoints. A chaque  tape ,
    on fusionne
 * les deux clusters les plus proches, avec une pond ration li e
```

```
    au poids des
 * clusters (cette distance est conserv e en m moire au cours des
    it rations).
 * Lors de la fusion de deux clusters, on choisit comme
    repr sentant un des deux
 * repr sentant des clsuters au hasard (   perfectionner) On
    parcours la liste et
 * la colonne de ce repr sentant, et on y met   jour la distance
    pour chaque
 * autre repr sentant de cluster.
 *
 * @author Louis Proffit
 * @version 1.0
 */
public class ClusteringHierarchical extends ClusteringSolver {

    private ArrayList<Checkpoint> checkpoints = new ArrayList<>();
    private ArrayList<Cluster> clusters = new ArrayList<>();

    public void addCluster(Cluster cluster, ImproveType improveType)
       {
        clusters.add(cluster);
        improve(improveType);
    }

    public void addCheckpoint(Checkpoint checkpoint, ImproveType
       improveType) {
        checkpoints.add(checkpoint);
        improve(improveType);
    }

    /**
     * Cherche et renvoie la plus petite distance entre deux
        clusters dans le
     * tableau distance. Cette distace est pond r e du nombre de
        checkpoints dans le
     * cluster.
     *
     * @return Un tableau d'indices sous la forme [i, j] ou i et j
        sont les indices
     *         des deux clusters les plus proches
     */
    private int[] getMinimumDistance(double[][] distances, int[]
       classes) {
        double min = 2;
        int xmin = -1;
        int ymin = -1;
        for (int i = 0; i < distances.length; i++) {
            for (int j = 0; j < i; j++) {
```

```
                if (distances[i][j] < min & classes[i] == i &
                    classes[j] == j) {
                    min = distances[i][j];
                    xmin = i;
                    ymin = j;
                }
            }
        }
        return new int[] { xmin, ymin };
    }

    /**
     * Mthode pour initialiser un tableau d'union find
     *
     * @param numberOfCheckpoints : La longueur du tableau
        d'union-find
     * @return un tableau de longueur <b>numberOfCheckpoints</b>
        contenant i
     *          l'index i.
     */
    private static int[] getClasses(int numberOfCheckpoints) {
        int[] classes = new int[numberOfCheckpoints];
        for (int i = 0; i < numberOfCheckpoints; i++)
            classes[i] = i;
        return classes;
    }

    /**
     * Mthode pour initialiser la matrice des distances
     *
     * @param checkpoints : La liste des checkpoints
     * @return Une matrice carre de c t  le nombre des
        checkpoints telle que
     *          M_{i,j} = d(C_i, C_j)
     */
    private double[][] getDistances() {
        int numberOfCheckpoints = checkpoints.size();
        double[][] distances = new
           double[numberOfCheckpoints][numberOfCheckpoints];
        double distance;
        for (int i = 0; i < numberOfCheckpoints; i++) {
            for (int j = 0; j < i; j++) {
                distance =
                   checkpoints.get(i).distance(checkpoints.get(j));
                distances[i][j] = distance;
                distances[j][i] = distance;
            }
        }
        return distances;
```

```
    }

    void mergeClusters(int firstClusterIndex, int
        secondClusterIndex, double[][] distances, int[] classes) {
        // Action on distances
        for (int i = 0; i < distances.length; i++) {
            distances[i][firstClusterIndex] =
                Math.max(distances[i][firstClusterIndex],
                    distances[i][secondClusterIndex]);
            distances[firstClusterIndex][i] =
                distances[i][firstClusterIndex];
        }
        // Action on classes
        assert (classes[firstClusterIndex] == firstClusterIndex);
        assert (classes[secondClusterIndex] == secondClusterIndex);
        classes[secondClusterIndex] = firstClusterIndex;
    }

    private void performClustering(int[] classes) {
        Cluster[] clustersAssociation = new Cluster[classes.length];
        int currentClusterIndex = 0;
        for (int i = 0; i < classes.length; i++) {
            if (classes[i] == i) {
                clustersAssociation[i] =
                    clusters.get(currentClusterIndex);
                currentClusterIndex++;
            }
        }
        assert (currentClusterIndex == clusters.size());
        for (int i = 0; i < classes.length; i++)
            clustersAssociation[findRepresentant(i,
                classes)].addCheckpoint(checkpoints.get(i));
    }

    private int findRepresentant(int index, int[] classes) {
        if (classes[index] == index)
            return index;
        return findRepresentant(classes[index], classes);
    }

    private void clearClusters() {
        for (Cluster cluster : clusters)
            cluster.clear();
    }

    @Override
    void improve(ImproveType improve) {
        if (improve == ImproveType.NULL)
            return;
```

```
        else if (improve == ImproveType.SIMPLE)
            return;
        clearClusters();
        int numberOfClusters = clusters.size();
        int numberOfCheckpoints = checkpoints.size();
        int[] classes = getClasses(numberOfCheckpoints);
        double[][] distances = getDistances();
        int mergesToPerform = numberOfCheckpoints - numberOfClusters;
        for (int i = 0; i < mergesToPerform; i++) {
            int[] clustersToMerge = getMinimumDistance(distances,
                classes);
            assert (clustersToMerge.length == 2);
            mergeClusters(clustersToMerge[0], clustersToMerge[1],
                distances, classes);
        }
        performClustering(classes);
        for (Cluster cluster : clusters)
            cluster.improvePath();
    }
}
```

```
package algorithms;

import java.util.HashMap;
import java.util.LinkedList;
import java.util.Map.Entry;

import structure.Checkpoint;
import structure.Cluster;

/**
 * Classe qui g re l'association entre checkpoints et clusters.
    Elle optimise
 * cette association gr ce    un algorithme k-means.
 *
 * @author Louis Proffit
 * @version 1.0
 */
public class ClusteringKMeans extends ClusteringSolver {

    /**
     * Liste des clusters
     */
    protected LinkedList<Cluster> availableClusters = new
        LinkedList<>();

    /**
     * Matrice d'association entre checkpoints et clusters
     */
```

```
protected HashMap<Checkpoint, Cluster> association = new
    HashMap<>();

/**
 * Ajoute un checkpoint
 *
 * @param checkpoint : Le checkpoint    ajouter
 * @param improve    : Le travail    effectuer    la fin
 */
public void addCheckpoint(Checkpoint checkpoint, ImproveType
    improve) {
    double distance;
    double minDistance = 2.0f;
    Cluster minCluster = null;
    for (Cluster cluster : availableClusters) {
        if (cluster.getSize() == 0) {
            cluster.addCheckpoint(checkpoint);
            association.put(checkpoint, cluster);
            improve(improve);
            return;
        }
        distance = cluster.distance(checkpoint);
        if (distance < minDistance) {
            minDistance = distance;
            minCluster = cluster;
        }
    }
    minCluster.addCheckpoint(checkpoint);
    association.put(checkpoint, minCluster);
    improve(improve);
}

/**
 * Ajoute un cluster
 *
 * @param cluster : Le cluster    ajouter
 * @param improve : Le travail    effectuer    la fin
 */
public void addCluster(Cluster cluster, ImproveType improve) {
    availableClusters.add(cluster);
    improve(improve);
}

/**
 * Effectue un travail d'am lioration
 *
 * @param improve : Le type de travail d'am lioration
    effectuer. Si c'est NULL
 *                : on ne fait rien. Si c'est SIMPLE : on
```

```
     am liore le chemin
 *                l'int rieur de chacun des clusters. Si c'est
    COMPLETE : on
 *                effectue un passage de k-means complet, puis
    on efectue le
 *                m me travail que SIMPLE.
 * @see ImproveType
 */
public void improve(ImproveType improve) {
    if (improve == ImproveType.NULL)
        return;
    else if (improve == ImproveType.SIMPLE) {
        for (Cluster clusterLocal : availableClusters)
            clusterLocal.improvePath();
        return;
    } else if (improve == ImproveType.COMPLETE) {
        boolean modified = true;
        while (modified)
            modified = improveOneStep();
        for (Cluster cluster : availableClusters)
            cluster.improvePath();
        return;
    }
}

/**
 * Effectue une   tape   de k-means
 *
 * @return
 */
private boolean improveOneStep() {
    HashMap<Checkpoint, Cluster> modifications = new HashMap<>();
    boolean modified = false;
    double distance;
    double currentMinDistance;
    Cluster cluster;
    Cluster minCluster;
    for (Checkpoint checkpoint : association.keySet()) {
        cluster = association.get(checkpoint);
        currentMinDistance = cluster.distance(checkpoint);
        minCluster = null;
        for (Cluster otherCluster : availableClusters) {
            distance = otherCluster.distance(checkpoint);
            if (distance < currentMinDistance) {
                currentMinDistance = distance;
                minCluster = otherCluster;
                modified = true;
            }
        }
```

```
            if (minCluster != null) {
                modifications.put(checkpoint, minCluster);
            }
        }
        for (Entry<Checkpoint, Cluster> entry :
           modifications.entrySet()) {
            Cluster oldCluster = association.put(entry.getKey(),
               entry.getValue()); // On ajoute la nouvelle
               association
            oldCluster.removeCheckpoint(entry.getKey()); // On
               enl ve le checkpoint de son ancien cluster
            entry.getValue().addCheckpoint(entry.getKey()); // On
               ajoute le checkpoint dans son nouveau cluster
        }
        return modified;
    }
}
```

```
package algorithms;

import structure.Checkpoint;
import structure.Cluster;

public abstract class ClusteringSolver {

    abstract void addCluster(Cluster cluster, ImproveType
       improveType);

    abstract void addCheckpoint(Checkpoint checkpoint, ImproveType
       improveType);

    /**
     * Effectue un travail d'am lioration
     *
     * @param improve : Le type de travail d'am lioration
        effectuer. Si c'est NULL
     *                : on ne fait rien. Si c'est SIMPLE : on
        am liore le chemin
     *                l'int rieur de chacun des clusters. Si c'est
       COMPLETE : on
     *                effectue un passage de k-means complet, puis
       on efectue le
     *                m me travail que SIMPLE.
     * @see ImproveType
     */
    abstract void improve(ImproveType improve);
}
```

```
package algorithms;
```

```java
/**
 * Enum ration des types d'am lioration possibles, avec trois
    niveaux.
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public enum ImproveType {
    NULL, SIMPLE, COMPLETE
}
```

```java
package algorithms;

import structure.Modification;

/**
 * Interface permettant l'application d'un recuit simul
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public interface RecuitInterface {

    /**
     * Renvoie la taille du probl me
     *
     * @return la taille du probl me
     */
    int getSize();

    /**
     * Renvoie une proposition de modification
     *
     * @return : la proposition
     */
    Modification modificationFunction();

    /**
     * Renvoie l'am lioration engendr e par une modification. Une
        am lioration
     * positive au sens litt ral sera positive au sens strict.
     *
     * @param modification : la modification      valuer
     * @return la valeur de l'am lioration
     */
    Double improvementFunction(Modification modification);
```

```
    /**
     * Applique la modification
     *
     * @param modification
     */
    void commitFunction(Modification modification);

}
```

```
package algorithms;

import structure.Modification;

/**
 * Classe impl mentant une m thode de recuit simul .
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public class TSPRecuit {

    /**
     * Type de d croissance de la temp rature
     */
    public static DecroissanceType temperatureDecroissanceType =
       DecroissanceType.N;

    /**
     * Valeur initiale de la temp rature
     */
    public static double temperatureInitialValue = 1;

    /**
     * Nombre d' tapes  de simulation
     */
    public static int numberOfSteps = 100;

    /**
     * Nombre d' tapes  de simulation sans changement
     */
    public static int numberOfStepsWithoutChange = 10;

    /**
     * M thode statique d'am lioration du chemin
     *
     * @param toImprove : la structure  am liorer
     */
```

```
public static void improvePath(RecuitInterface toImprove) {
    int size = toImprove.getSize();
    if (size <= 3)
        return;
    int currentSteps = 0;
    int currentStepsWithoutChange = 0;
    double currentTemperature;
    double improvement;
    Modification modification;
    numberOfSteps = 1000 * size;
    numberOfStepsWithoutChange = 5 * size;

    while (currentSteps < numberOfSteps &
       currentStepsWithoutChange < numberOfStepsWithoutChange) {
        currentTemperature = temperature(currentSteps);
        modification = toImprove.modificationFunction();
        improvement =
           toImprove.improvementFunction(modification);
        if (h(Math.exp(-improvement / currentTemperature)) >
           Math.random()) {
            toImprove.commitFunction(modification);
            currentStepsWithoutChange = 0;
        } else
            currentStepsWithoutChange += 1;
        currentSteps++;
    }
}

/**
 * Fonction de temp rature, auxiliaire pour le recuit simul
 *
 * @param time : Le temps auquel on calcule la temp rature
 * @return La temp rature au temps <b>time</b>
 */
private static double temperature(int time) {
    switch (temperatureDecroissanceType) {
    case LOG:
        return temperatureInitialValue / Math.log(time);
    case N:
        return temperatureInitialValue / time;
    case N2:
        return temperatureInitialValue / (time * time);
    }
    return 0;
}

/**
 * Une fonction auxiliaire h pour le recuit simul , v rifie
    l' quation   h(x) =
```

```
     * xh(1/x)
     */
    private static double h(double x) {
        return x / (1 + x);
    }

    /**
     * Enum ration des types de d croissance possible de la
        temp rature
     */
    public static enum DecroissanceType {
        /**
         * D croissance logarithmique
    s     */
        LOG,

        /**
         * D croissance lin aire
         */
        N,

        /**
         * D croissance quadratique
         */
        N2
    }
}
```

1.2 Package graphics

```
package graphics;

import java.util.concurrent.TimeUnit;

import algorithms.ClusterAttribution;
import algorithms.ImproveType;
import structure.Checkpoint;
import structure.Drone;
import structure.MutableVector;

/**
 * Classe principale,    executer pour faire fonctionner les
    algorithmes.
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
```

```
 */
public class Controller {

    /**
     * Nombre d' tapes   de la simulation
     */
    private static int numberOfSteps = 1000;

    /**
     * Nombre de drones    g n rer au d part
     */
    private static final int numberOfDrones = 5;

    /**
     * Nombre de checkpoints    g n rer au d part
     */
    private static final int numberOfCheckpoints = 20;

    /**
     * Objet de calcul de l'attribuion des drones aux clusters
     */
    private static final ClusterAttribution clusterAttribution = new
       ClusterAttribution();

    /**
     * Objet graphique
     */
    private static final GraphicsInterface graphics = new
       LocalGraphics(1000, 1000);

    /**
     * Cr e une configuration initiale al atoire avec un nombre de
        drones et de
     * clusters fix . Am liore imm diatement la solution, et
        effectue un affichage
     */
    private static void init() {
        for (int i = 0; i < numberOfDrones; i++) {
            addDrone(new Drone(new MutableVector()),
               ImproveType.NULL);
        }
        for (int i = 0; i < numberOfCheckpoints; i++) {
            addCheckpoint(new Checkpoint(new MutableVector()),
               ImproveType.NULL);
        }
        clusterAttribution.improve(ImproveType.COMPLETE);
        draw();
    }
```

```
    /**
     * Met    jour la situation : les drones bougent et le graphique
          volue
     */
    private static void update() {
        clusterAttribution.move();
        draw();
    }

    /**
     * Met    jour le graphique
     */
    private static void draw() {
        graphics.updateGraphics(clusterAttribution);
    }

    /**
     * Ajoute un drone
     *
     * @param drone   : le drone    ajouter
     * @param improve : le travail    effectuer    l'issue
     */
    public static void addDrone(Drone drone, ImproveType improve) {
        clusterAttribution.addDrone(drone, improve);
    }

    /**
     * Ajoute un checkpoint
     *
     * @param checkpoint : le checkpoint    ajouter
     * @param improve    : le travail    effectuer    la fin
     */
    public static void addCheckpoint(Checkpoint checkpoint,
       ImproveType improve) {
        clusterAttribution.addCheckpoint(checkpoint, improve);
    }

    /**
     * Fait tourner la simulation
     */
    private static void run() {
        try {
            for (int i = 0; i < numberOfSteps; i++) {
                TimeUnit.MILLISECONDS.sleep(200);
                update();
            }
        } catch (InterruptedException e) {
            System.out.println("Simulation interrompue");
            e.printStackTrace();
```

```
        }

    }

    /**
     * Initialise et fait tourner la simulation
     */
    public static void main(String[] args) {
        init();
        run();
    }
}
```

```
package graphics;

import java.awt.BasicStroke;
import java.awt.Color;
import java.awt.Component;
import java.awt.FileDialog;
import java.awt.Font;
import java.awt.FontMetrics;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.MediaTracker;
import java.awt.RenderingHints;
import java.awt.Toolkit;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.awt.event.MouseMotionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;

import java.awt.geom.Arc2D;
import java.awt.geom.Ellipse2D;
import java.awt.geom.GeneralPath;
import java.awt.geom.Line2D;
import java.awt.geom.Rectangle2D;

import java.awt.image.BufferedImage;
import java.awt.image.DirectColorModel;
import java.awt.image.WritableRaster;

import java.io.File;
import java.io.IOException;
```

```
import java.net.MalformedURLException;
import java.net.URL;

import java.util.ArrayList;
import java.util.LinkedList;
import java.util.TreeSet;

import javax.imageio.ImageIO;

import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JMenu;
import javax.swing.JMenuBar;
import javax.swing.JMenuItem;
import javax.swing.KeyStroke;

/**
 * <i>Draw</i>.  This class provides a basic capability for creating
    drawings
 * with your programs.  It uses a simple graphics model that allows
    you to create
 * drawings consisting of points, lines, and curves in a window on
    your computer
 * and to save the drawings to a file.  This is the object-oriented
    version of
 * standard draw; it supports multiple indepedent drawing windows.
 * <p>
 * For additional documentation, see
 * <a href="https://introcs.cs.princeton.edu/31datatype">Section
    3.1</a> of
 * <i>Computer Science: An Interdisciplinary Approach</i> by Robert
    Sedgewick
 * and Kevin Wayne.
 *
 * @author Robert Sedgewick
 * @author Kevin Wayne
 */
public final class Draw implements ActionListener, MouseListener,
   MouseMotionListener, KeyListener {

    /**
     * The color black.
     */
    public static final Color BLACK = Color.BLACK;

    /**
     * The color blue.
     */
```

```
    public static final Color BLUE = Color.BLUE;

    /**
     * The color cyan.
     */
    public static final Color CYAN = Color.CYAN;

    /**
     * The color dark gray.
     */
    public static final Color DARK_GRAY = Color.DARK_GRAY;

    /**
     * The color gray.
     */
    public static final Color GRAY = Color.GRAY;

    /**
     * The color green.
     */
    public static final Color GREEN = Color.GREEN;

    /**
     * The color light gray.
     */
    public static final Color LIGHT_GRAY = Color.LIGHT_GRAY;

    /**
     * The color magenta.
     */
    public static final Color MAGENTA = Color.MAGENTA;

    /**
     * The color orange.
     */
    public static final Color ORANGE = Color.ORANGE;

    /**
     * The color pink.
     */
    public static final Color PINK = Color.PINK;

    /**
     * The color red.
     */
    public static final Color RED = Color.RED;

    /**
     * The color white.
```

```
     */
    public static final Color WHITE = Color.WHITE;

    /**
     * The color yellow.
     */
    public static final Color YELLOW = Color.YELLOW;

    /**
     * Shade of blue used in Introduction to Programming in Java. It
        is Pantone
     * 300U. The RGB values are approximately (9, 90, 166).
     */
    public static final Color BOOK_BLUE = new Color(9, 90, 166);

    /**
     * Shade of light blue used in Introduction to Programming in
        Java. The RGB
     * values are approximately (103, 198, 243).
     */
    public static final Color BOOK_LIGHT_BLUE = new Color(103, 198,
       243);

    /**
     * Shade of red used in <em>Algorithms, 4th edition</em>. It is
        Pantone 1805U.
     * The RGB values are approximately (150, 35, 31).
     */
    public static final Color BOOK_RED = new Color(150, 35, 31);

    /**
     * Shade of orange used in Princeton's identity. It is PMS 158.
        The RGB values
     * are approximately (245, 128, 37).
     */
    public static final Color PRINCETON_ORANGE = new Color(245, 128,
       37);

    // default colors
    private static final Color DEFAULT_PEN_COLOR = BLACK;
    private static final Color DEFAULT_CLEAR_COLOR = WHITE;

    // boundary of drawing canvas, 0% border
    private static final double BORDER = 0.0;
    private static final double DEFAULT_XMIN = 0.0;
    private static final double DEFAULT_XMAX = 1.0;
    private static final double DEFAULT_YMIN = 0.0;
    private static final double DEFAULT_YMAX = 1.0;
```

```
// default canvas size is SIZE-by-SIZE
private static final int DEFAULT_SIZE = 512;

// default pen radius
private static final double DEFAULT_PEN_RADIUS = 0.002;

// default font
private static final Font DEFAULT_FONT = new Font("SansSerif",
   Font.PLAIN, 16);

// current pen color
private Color penColor;

// canvas size
private int width = DEFAULT_SIZE;
private int height = DEFAULT_SIZE;

// current pen radius
private double penRadius;

// show we draw immediately or wait until next show?
private boolean defer = false;

private double xmin, ymin, xmax, ymax;

// name of window
private String name = "Draw";

// for synchronization
private final Object mouseLock = new Object();
private final Object keyLock = new Object();

// current font
private Font font;

// the JLabel for drawing
private JLabel draw;

// double buffered graphics
private BufferedImage offscreenImage, onscreenImage;
private Graphics2D offscreen, onscreen;

// the frame for drawing to the screen
private JFrame frame = new JFrame();

// mouse state
private boolean isMousePressed = false;
private double mouseX = 0;
private double mouseY = 0;
```

```
// keyboard state
private final LinkedList<Character> keysTyped = new
   LinkedList<Character>();
private final TreeSet<Integer> keysDown = new TreeSet<Integer>();

// event-based listeners
private final ArrayList<DrawListener> listeners = new
   ArrayList<DrawListener>();

/**
 * Initializes an empty drawing object with the given name.
 *
 * @param name the title of the drawing window.
 */
public Draw(String name) {
    this.name = name;
    init();
}

/**
 * Initializes an empty drawing object.
 */
public Draw() {
    init();
}

private void init() {
    if (frame != null)
        frame.setVisible(false);
    frame = new JFrame();
    // Ligne rajout e pour  tre  en echelle pleine
    offscreenImage = new BufferedImage(width, height,
       BufferedImage.TYPE_INT_ARGB);
    onscreenImage = new BufferedImage(width, height,
       BufferedImage.TYPE_INT_ARGB);
    offscreen = offscreenImage.createGraphics();
    onscreen = onscreenImage.createGraphics();

    setXscale();
    setYscale();
    offscreen.setColor(DEFAULT_CLEAR_COLOR);
    offscreen.fillRect(0, 0, width, height);
    setPenColor();
    setPenRadius();
    setFont();
    clear();

    // add antialiasing
```

```
        RenderingHints hints = new
            RenderingHints(RenderingHints.KEY_ANTIALIASING,
            RenderingHints.VALUE_ANTIALIAS_ON);
        hints.put(RenderingHints.KEY_RENDERING,
            RenderingHints.VALUE_RENDER_QUALITY);
        offscreen.addRenderingHints(hints);

        // frame stuff
        RetinaImageIcon icon = new RetinaImageIcon(onscreenImage);
        draw = new JLabel(icon);

        draw.addMouseListener(this);
        draw.addMouseMotionListener(this);

        frame.setContentPane(draw);
        frame.addKeyListener(this); // JLabel cannot get keyboard
            focus
        frame.setResizable(false);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE); //
            closes all windows
        frame.setDefaultCloseOperation(JFrame.DISPOSE_ON_CLOSE); //
            closes only current window
        frame.setFocusTraversalKeysEnabled(false); // to recognize
            VK_TAB with isKeyPressed()
        frame.setTitle(name);
        frame.setJMenuBar(createMenuBar());
        frame.pack();
        frame.requestFocusInWindow();
        frame.setVisible(true);
    }

    /**
     * Sets the upper-left hand corner of the drawing window to be
        (x, y), where (0,
     * 0) is upper left.
     *
     * @param x the number of pixels from the left
     * @param y the number of pixels from the top
     * @throws IllegalArgumentException if the width or height is 0
        or negative
     */
    public void setLocationOnScreen(int x, int y) {
        if (x <= 0 || y <= 0)
            throw new IllegalArgumentException();
        frame.setLocation(x, y);
    }

    /**
     * Sets the default close operation.
```

```
     *
     * @param value the value, typically {@code
        JFrame.EXIT_ON_CLOSE} (close all
     *              windows) or {@code JFrame.DISPOSE_ON_CLOSE}
        (close current
     *              window)
     */
    public void setDefaultCloseOperation(int value) {
        frame.setDefaultCloseOperation(value);
    }

    /**
     * Sets the canvas (drawing area) to be
        <em>width</em>-by-<em>height</em>
     * pixels. This also erases the current drawing and resets the
        coordinate
     * system, pen radius, pen color, and font back to their default
        values.
     * Ordinarly, this method is called once, at the very beginning
        of a program.
     *
     * @param canvasWidth  the width as a number of pixels
     * @param canvasHeight the height as a number of pixels
     * @throws IllegalArgumentException unless both {@code
        canvasWidth} and
     *                                  {@code canvasHeight} are
        positive
     */
    public void setCanvasSize(int canvasWidth, int canvasHeight) {
        if (canvasWidth < 1 || canvasHeight < 1) {
            throw new IllegalArgumentException("width␣and␣height␣
                must␣be␣positive");
        }
        width = canvasWidth;
        height = canvasHeight;
        init();
    }

    // create the menu bar (changed to private)
    @SuppressWarnings("deprecation")
    private JMenuBar createMenuBar() {
        JMenuBar menuBar = new JMenuBar();
        JMenu menu = new JMenu("File");
        menuBar.add(menu);
        JMenuItem menuItem1 = new JMenuItem("␣Save...␣␣␣");
        menuItem1.addActionListener(this);
        // Java 10+: replace getMenuShortcutKeyMask() with
           getMenuShortcutKeyMaskEx()
        menuItem1.setAccelerator(
```

```
            KeyStroke.getKeyStroke(KeyEvent.VK_S,
               Toolkit.getDefaultToolkit().getMenuShortcutKeyMask()));
        menu.add(menuItem1);
        return menuBar;
    }

    /* ***************************************************************************
     * User and screen coordinate systems.
     *************************************************************************** */

    // throw an IllegalArgumentException if x is NaN or infinite
    private static void validate(double x, String name) {
        if (Double.isNaN(x))
            throw new IllegalArgumentException(name + " is NaN");
        if (Double.isInfinite(x))
            throw new IllegalArgumentException(name + " is
               infinite");
    }

    // throw an IllegalArgumentException if s is null
    private static void validateNonnegative(double x, String name) {
        if (x < 0)
            throw new IllegalArgumentException(name + " negative");
    }

    // throw an IllegalArgumentException if s is null
    private static void validateNotNull(Object x, String name) {
        if (x == null)
            throw new IllegalArgumentException(name + " is null");
    }

    /**
     * Sets the x-scale to be the default (between 0.0 and 1.0).
     */
    public void setXscale() {
        setXscale(DEFAULT_XMIN, DEFAULT_XMAX);
    }

    /**
     * Sets the y-scale to be the default (between 0.0 and 1.0).
     */
    public void setYscale() {
        setYscale(DEFAULT_YMIN, DEFAULT_YMAX);
    }

    /**
     * Sets the x-scale.
     *
     * @param min the minimum value of the x-scale
```

```
     * @param max the maximum value of the x-scale
     * @throws IllegalArgumentException if {@code (max == min)}
     * @throws IllegalArgumentException if either {@code min} or
        {@code max} is
     *                                  either NaN or infinite
     */
    public void setXscale(double min, double max) {
        validate(min, "min");
        validate(max, "max");
        double size = max - min;
        if (size == 0.0)
            throw new IllegalArgumentException("the min and max are
                the same");
        xmin = min - BORDER * size;
        xmax = max + BORDER * size;
    }

    /**
     * Sets the y-scale.
     *
     * @param min the minimum value of the y-scale
     * @param max the maximum value of the y-scale
     * @throws IllegalArgumentException if {@code (max == min)}
     * @throws IllegalArgumentException if either {@code min} or
        {@code max} is
     *                                  either NaN or infinite
     */
    public void setYscale(double min, double max) {
        validate(min, "min");
        validate(max, "max");
        double size = max - min;
        if (size == 0.0)
            throw new IllegalArgumentException("the min and max are
                the same");
        ymin = min - BORDER * size;
        ymax = max + BORDER * size;
    }

    // helper functions that scale from user coordinates to screen
       coordinates and
    // back
    private double scaleX(double x) {
        return width * (x - xmin) / (xmax - xmin);
    }

    private double scaleY(double y) {
        return height * (ymax - y) / (ymax - ymin);
    }
```

```
    private double factorX(double w) {
        return w * width / Math.abs(xmax - xmin);
    }

    private double factorY(double h) {
        return h * height / Math.abs(ymax - ymin);
    }

    private double userX(double x) {
        return xmin + x * (xmax - xmin) / width;
    }

    private double userY(double y) {
        return ymax - y * (ymax - ymin) / height;
    }

    /**
     * Clears the screen to the default color (white).
     */
    public void clear() {
        clear(DEFAULT_CLEAR_COLOR);
    }

    /**
     * Clears the screen to the given color.
     *
     * @param color the color to make the background
     * @throws IllegalArgumentException if {@code color} is {@code
        null}
     */
    public void clear(Color color) {
        validateNotNull(color, "color");
        offscreen.setColor(color);
        offscreen.fillRect(0, 0, width, height);
        offscreen.setColor(penColor);
        draw();
    }

    /**
     * Gets the current pen radius.
     *
     * @return the current pen radius
     */
    public double getPenRadius() {
        return penRadius;
    }

    /**
     * Sets the pen size to the default (.002).
```

```
     */
    public void setPenRadius() {
        setPenRadius(DEFAULT_PEN_RADIUS);
    }

    /**
     * Sets the radius of the pen to the given size.
     *
     * @param radius the radius of the pen
     * @throws IllegalArgumentException if {@code radius} is
         negative, NaN, or
     *                                  infinite
     */
    public void setPenRadius(double radius) {
        validate(radius, "pen radius");
        validateNonnegative(radius, "pen radius");

        penRadius = radius * DEFAULT_SIZE;
        BasicStroke stroke = new BasicStroke((float) penRadius,
            BasicStroke.CAP_ROUND, BasicStroke.JOIN_ROUND);
        // BasicStroke stroke = new BasicStroke((float) penRadius);
        offscreen.setStroke(stroke);
    }

    /**
     * Gets the current pen color.
     *
     * @return the current pen color
     */
    public Color getPenColor() {
        return penColor;
    }

    /**
     * Sets the pen color to the default color (black).
     */
    public void setPenColor() {
        setPenColor(DEFAULT_PEN_COLOR);
    }

    /**
     * Sets the pen color to the given color.
     *
     * @param color the color to make the pen
     * @throws IllegalArgumentException if {@code color} is {@code
         null}
     */
    public void setPenColor(Color color) {
        validateNotNull(color, "color");
```

```
        penColor = color;
        offscreen.setColor(penColor);
    }

    /**
     * Sets the pen color to the given RGB color.
     *
     * @param red   the amount of red (between 0 and 255)
     * @param green the amount of green (between 0 and 255)
     * @param blue  the amount of blue (between 0 and 255)
     * @throws IllegalArgumentException if {@code red}, {@code
        green}, or
     *                                  {@code blue} is outside its
        prescribed range
     */
    public void setPenColor(int red, int green, int blue) {
        if (red < 0 || red >= 256)
            throw new IllegalArgumentException("red must be between
                0 and 255");
        if (green < 0 || green >= 256)
            throw new IllegalArgumentException("green must be
                between 0 and 255");
        if (blue < 0 || blue >= 256)
            throw new IllegalArgumentException("blue must be between
                0 and 255");
        setPenColor(new Color(red, green, blue));
    }

    /**
     * Turns on xor mode.
     */
    public void xorOn() {
        offscreen.setXORMode(DEFAULT_CLEAR_COLOR);
    }

    /**
     * Turns off xor mode.
     */
    public void xorOff() {
        offscreen.setPaintMode();
    }

    /**
     * Gets the current {@code JLabel} for use in some other GUI.
     *
     * @return the current {@code JLabel}
     */
    public JLabel getJLabel() {
        return draw;
```

```
    }

    /**
     * Gets the current font.
     *
     * @return the current font
     */
    public Font getFont() {
        return font;
    }

    /**
     * Sets the font to the default font (sans serif, 16 point).
     */
    public void setFont() {
        setFont(DEFAULT_FONT);
    }

    /**
     * Sets the font to the given value.
     *
     * @param font the font
     * @throws IllegalArgumentException if {@code font} is {@code
         null}
     */
    public void setFont(Font font) {
        validateNotNull(font, "font");
        this.font = font;
    }

    /* ***************************************************************************
     * Drawing geometric shapes.
     *************************************************************************** */

    /**
     * Draws a line from (x0, y0) to (x1, y1).
     *
     * @param x0 the x-coordinate of the starting point
     * @param y0 the y-coordinate of the starting point
     * @param x1 the x-coordinate of the destination point
     * @param y1 the y-coordinate of the destination point
     * @throws IllegalArgumentException if any coordinate is either
         NaN or infinite
     */
    public void line(double x0, double y0, double x1, double y1) {
        validate(x0, "x0");
        validate(y0, "y0");
        validate(x1, "x1");
        validate(y1, "y1");
```

```
        offscreen.draw(new Line2D.Double(scaleX(x0), scaleY(y0),
           scaleX(x1), scaleY(y1)));
        draw();
    }

    /**
     * Draws one pixel at (x, y).
     *
     * @param x the x-coordinate of the pixel
     * @param y the y-coordinate of the pixel
     * @throws IllegalArgumentException if {@code x} or {@code y} is
        either NaN or
     *                                  infinite
     */
    private void pixel(double x, double y) {
        validate(x, "x");
        validate(y, "y");
        offscreen.fillRect((int) Math.round(scaleX(x)), (int)
           Math.round(scaleY(y)), 1, 1);
    }

    /**
     * Draws a point at (x, y).
     *
     * @param x the x-coordinate of the point
     * @param y the y-coordinate of the point
     * @throws IllegalArgumentException if either {@code x} or
        {@code y} is either
     *                                  NaN or infinite
     */
    public void point(double x, double y) {
        validate(x, "x");
        validate(y, "y");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double r = penRadius;
        // double ws = factorX(2*r);
        // double hs = factorY(2*r);
        // if (ws <= 1 && hs <= 1) pixel(x, y);
        if (r <= 1)
            pixel(x, y);
        else
            offscreen.fill(new Ellipse2D.Double(xs - r / 2, ys - r /
               2, r, r));
        draw();
    }

    /**
```

```
     * Draws a circle of the specified radius, centered at
       (<em>x</em>, <em>y</em>).
     *
     * @param x      the x-coordinate of the center of the circle
     * @param y      the y-coordinate of the center of the circle
     * @param radius the radius of the circle
     * @throws IllegalArgumentException if {@code radius} is negative
     * @throws IllegalArgumentException if any argument is either
       NaN or infinite
     */
    public void circle(double x, double y, double radius) {
        validate(x, "x");
        validate(y, "y");
        validate(radius, "radius");
        validateNonnegative(radius, "radius");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2 * radius);
        double hs = factorY(2 * radius);
        if (ws <= 1 && hs <= 1)
            pixel(x, y);
        else
            offscreen.draw(new Ellipse2D.Double(xs - ws / 2, ys - hs
               / 2, ws, hs));
        draw();
    }

    /**
     * Draws a filled circle of the specified radius, centered at
       (<em>x</em>,
     * <em>y</em>).
     *
     * @param x      the x-coordinate of the center of the circle
     * @param y      the y-coordinate of the center of the circle
     * @param radius the radius of the circle
     * @throws IllegalArgumentException if {@code radius} is negative
     * @throws IllegalArgumentException if any argument is either
       NaN or infinite
     */
    public void filledCircle(double x, double y, double radius) {
        validate(x, "x");
        validate(y, "y");
        validate(radius, "radius");
        validateNonnegative(radius, "radius");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2 * radius);
```

```
        double hs = factorY(2 * radius);
        if (ws <= 1 && hs <= 1)
            pixel(x, y);
        else
            offscreen.fill(new Ellipse2D.Double(xs - ws / 2, ys - hs
                / 2, ws, hs));
        draw();
    }

    /**
     * Draws an ellipse with the specified semimajor and semiminor
        axes, centered at
     * (<em>x</em>, <em>y</em>).
     *
     * @param x              the <em>x</em>-coordinate of the center
        of the ellipse
     * @param y              the <em>y</em>-coordinate of the center
        of the ellipse
     * @param semiMajorAxis is the semimajor axis of the ellipse
     * @param semiMinorAxis is the semiminor axis of the ellipse
     * @throws IllegalArgumentException if either {@code
        semiMajorAxis} or
     *                                  {@code semiMinorAxis} is
        negative
     * @throws IllegalArgumentException if any argument is either
        NaN or infinite
     */
    public void ellipse(double x, double y, double semiMajorAxis,
       double semiMinorAxis) {
        validate(x, "x");
        validate(y, "y");
        validate(semiMajorAxis, "semimajor axis");
        validate(semiMinorAxis, "semiminor axis");
        validateNonnegative(semiMajorAxis, "semimajor axis");
        validateNonnegative(semiMinorAxis, "semiminor axis");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2 * semiMajorAxis);
        double hs = factorY(2 * semiMinorAxis);
        if (ws <= 1 && hs <= 1)
            pixel(x, y);
        else
            offscreen.draw(new Ellipse2D.Double(xs - ws / 2, ys - hs
                / 2, ws, hs));
        draw();
    }

    /**
```

```
     * Draws a filled ellipse with the specified semimajor and
        semiminor axes,
     * centered at (<em>x</em>, <em>y</em>).
     *
     * @param x              the <em>x</em>-coordinate of the center
        of the ellipse
     * @param y              the <em>y</em>-coordinate of the center
        of the ellipse
     * @param semiMajorAxis is the semimajor axis of the ellipse
     * @param semiMinorAxis is the semiminor axis of the ellipse
     * @throws IllegalArgumentException if either {@code
        semiMajorAxis} or
     *                                  {@code semiMinorAxis} is
        negative
     * @throws IllegalArgumentException if any argument is either
        NaN or infinite
     */
    public void filledEllipse(double x, double y, double
        semiMajorAxis, double semiMinorAxis) {
        validate(x, "x");
        validate(y, "y");
        validate(semiMajorAxis, "semimajor axis");
        validate(semiMinorAxis, "semiminor axis");
        validateNonnegative(semiMajorAxis, "semimajor axis");
        validateNonnegative(semiMinorAxis, "semiminor axis");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2 * semiMajorAxis);
        double hs = factorY(2 * semiMinorAxis);
        if (ws <= 1 && hs <= 1)
            pixel(x, y);
        else
            offscreen.fill(new Ellipse2D.Double(xs - ws / 2, ys - hs
                / 2, ws, hs));
        draw();
    }

    /**
     * Draws a circular arc of the specified radius, centered at
        (<em>x</em>,
     * <em>y</em>), from angle1 to angle2 (in degrees).
     *
     * @param x      the <em>x</em>-coordinate of the center of the
        circle
     * @param y      the <em>y</em>-coordinate of the center of the
        circle
     * @param radius the radius of the circle
     * @param angle1 the starting angle. 0 would mean an arc
```

```
     beginning at 3 o'clock.
  * @param angle2 the angle at the end of the arc. For example,
     if you want a 90
  *               degree arc, then angle2 should be angle1 + 90.
  * @throws IllegalArgumentException if {@code radius} is negative
  * @throws IllegalArgumentException if any argument is either
     NaN or infinite
  */
 public void arc(double x, double y, double radius, double
    angle1, double angle2) {
     validate(x, "x");
     validate(y, "y");
     validate(radius, "arc radius");
     validate(angle1, "angle1");
     validate(angle2, "angle2");
     validateNonnegative(radius, "arc radius");

     while (angle2 < angle1)
         angle2 += 360;
     double xs = scaleX(x);
     double ys = scaleY(y);
     double ws = factorX(2 * radius);
     double hs = factorY(2 * radius);
     if (ws <= 1 && hs <= 1)
         pixel(x, y);
     else
         offscreen.draw(new Arc2D.Double(xs - ws / 2, ys - hs /
            2, ws, hs, angle1, angle2 - angle1, Arc2D.OPEN));
     draw();
 }

 /**
  * Draws a square of the specified size, centered at
     (<em>x</em>, <em>y</em>).
  *
  * @param x          the <em>x</em>-coordinate of the center of
     the square
  * @param y          the <em>y</em>-coordinate of the center of
     the square
  * @param halfLength one half the length of any side of the
     square
  * @throws IllegalArgumentException if {@code halfLength} is
     negative
  * @throws IllegalArgumentException if any argument is either
     NaN or infinite
  */
 public void square(double x, double y, double halfLength) {
     validate(x, "x");
     validate(y, "y");
```

```
        validate(halfLength, "halfLength");
        validateNonnegative(halfLength, "half length");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2 * halfLength);
        double hs = factorY(2 * halfLength);
        if (ws <= 1 && hs <= 1)
            pixel(x, y);
        else
            offscreen.draw(new Rectangle2D.Double(xs - ws / 2, ys -
               hs / 2, ws, hs));
        draw();
    }

    /**
     * Draws a square of the specified size, centered at
        (<em>x</em>, <em>y</em>).
     *
     * @param x           the <em>x</em>-coordinate of the center of
        the square
     * @param y           the <em>y</em>-coordinate of the center of
        the square
     * @param halfLength one half the length of any side of the
        square
     * @throws IllegalArgumentException if {@code halfLength} is
        negative
     * @throws IllegalArgumentException if any argument is either
        NaN or infinite
     */
    public void filledSquare(double x, double y, double halfLength) {
        validate(x, "x");
        validate(y, "y");
        validate(halfLength, "halfLength");
        validateNonnegative(halfLength, "half length");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2 * halfLength);
        double hs = factorY(2 * halfLength);
        if (ws <= 1 && hs <= 1)
            pixel(x, y);
        else
            offscreen.fill(new Rectangle2D.Double(xs - ws / 2, ys -
               hs / 2, ws, hs));
        draw();
    }

    /**
```

```
     * Draws a rectangle of the specified size, centered at
       (<em>x</em>,
     * <em>y</em>).
     *
     * @param x           the <em>x</em>-coordinate of the center of
       the rectangle
     * @param y           the <em>y</em>-coordinate of the center of
       the rectangle
     * @param halfWidth   one half the width of the rectangle
     * @param halfHeight one half the height of the rectangle
     * @throws IllegalArgumentException if either {@code halfWidth}
       or
     *                                  {@code halfHeight} is
       negative
     * @throws IllegalArgumentException if any argument is either
       NaN or infinite
     */
    public void rectangle(double x, double y, double halfWidth,
       double halfHeight) {
        validate(x, "x");
        validate(y, "y");
        validate(halfWidth, "halfWidth");
        validate(halfHeight, "halfHeight");
        validateNonnegative(halfWidth, "half␣width");
        validateNonnegative(halfHeight, "half␣height");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2 * halfWidth);
        double hs = factorY(2 * halfHeight);
        if (ws <= 1 && hs <= 1)
            pixel(x, y);
        else
            offscreen.draw(new Rectangle2D.Double(xs - ws / 2, ys -
               hs / 2, ws, hs));
        draw();
    }

    /**
     * Draws a filled rectangle of the specified size, centered at
       (<em>x</em>,
     * <em>y</em>).
     *
     * @param x           the <em>x</em>-coordinate of the center of
       the rectangle
     * @param y           the <em>y</em>-coordinate of the center of
       the rectangle
     * @param halfWidth   one half the width of the rectangle
     * @param halfHeight one half the height of the rectangle
```

```
     * @throws IllegalArgumentException if either {@code halfWidth}
        or
     *                                  {@code halfHeight} is
        negative
     * @throws IllegalArgumentException if any argument is either
        NaN or infinite
     */
    public void filledRectangle(double x, double y, double
       halfWidth, double halfHeight) {
        validate(x, "x");
        validate(y, "y");
        validate(halfWidth, "halfWidth");
        validate(halfHeight, "halfHeight");
        validateNonnegative(halfWidth, "half width");
        validateNonnegative(halfHeight, "half height");

        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(2 * halfWidth);
        double hs = factorY(2 * halfHeight);
        if (ws <= 1 && hs <= 1)
            pixel(x, y);
        else
            offscreen.fill(new Rectangle2D.Double(xs - ws / 2, ys -
               hs / 2, ws, hs));
        draw();
    }

    /**
     * Draws a polygon with the vertices (<em>x</em><sub>0</sub>,
     * <em>y</em><sub>0</sub>), (<em>x</em><sub>1</sub>,
        <em>y</em><sub>1</sub>),
     * ..., (<em>x</em><sub><em>n</em>     1     </sub>,
     * <em>y</em><sub><em>n</em>     1     </sub>).
     *
     * @param x an array of all the <em>x</em>-coordinates of the
        polygon
     * @param y an array of all the <em>y</em>-coordinates of the
        polygon
     * @throws IllegalArgumentException unless {@code x[]} and
        {@code y[]} are of
     *                                  the same length
     * @throws IllegalArgumentException if any coordinate is either
        NaN or infinite
     * @throws IllegalArgumentException if either {@code x[]} or
        {@code y[]} is
     *                                  {@code null}
     */
    public void polygon(double[] x, double[] y) {
```

```
        validateNotNull(x, "x-coordinate array");
        validateNotNull(y, "y-coordinate array");
        for (int i = 0; i < x.length; i++)
            validate(x[i], "x[" + i + "]");
        for (int i = 0; i < y.length; i++)
            validate(y[i], "y[" + i + "]");

        int n1 = x.length;
        int n2 = y.length;
        if (n1 != n2)
            throw new IllegalArgumentException("arrays must be of
                the same length");
        int n = n1;
        if (n == 0)
            return;

        GeneralPath path = new GeneralPath();
        path.moveTo((float) scaleX(x[0]), (float) scaleY(y[0]));
        for (int i = 0; i < n; i++)
            path.lineTo((float) scaleX(x[i]), (float) scaleY(y[i]));
        path.closePath();
        offscreen.draw(path);
        draw();
    }

    /**
     * Draws a filled polygon with the vertices
        (<em>x</em><sub>0</sub>,
     * <em>y</em><sub>0</sub>), (<em>x</em><sub>1</sub>,
        <em>y</em><sub>1</sub>),
     * ..., (<em>x</em><sub><em>n</em>  1  </sub>,
     * <em>y</em><sub><em>n</em>  1  </sub>).
     *
     * @param x an array of all the <em>x</em>-coordinates of the
        polygon
     * @param y an array of all the <em>y</em>-coordinates of the
        polygon
     * @throws IllegalArgumentException unless {@code x[]} and
        {@code y[]} are of
     *                                  the same length
     * @throws IllegalArgumentException if any coordinate is either
        NaN or infinite
     * @throws IllegalArgumentException if either {@code x[]} or
        {@code y[]} is
     *                                  {@code null}
     */
    public void filledPolygon(double[] x, double[] y) {
        validateNotNull(x, "x-coordinate array");
        validateNotNull(y, "y-coordinate array");
```

```
        for (int i = 0; i < x.length; i++)
            validate(x[i], "x[" + i + "]");
        for (int i = 0; i < y.length; i++)
            validate(y[i], "y[" + i + "]");

        int n1 = x.length;
        int n2 = y.length;
        if (n1 != n2)
            throw new IllegalArgumentException("arrays must be of
               the same length");
        int n = n1;
        if (n == 0)
            return;

        GeneralPath path = new GeneralPath();
        path.moveTo((float) scaleX(x[0]), (float) scaleY(y[0]));
        for (int i = 0; i < n; i++)
            path.lineTo((float) scaleX(x[i]), (float) scaleY(y[i]));
        path.closePath();
        offscreen.fill(path);
        draw();
    }

    /* ***********************************************************************
     * Drawing images.
     *********************************************************************** */

    // get an image from the given filename
    private static Image getImage(String filename) {
        if (filename == null)
            throw new IllegalArgumentException();

        // to read from file
        ImageIcon icon = new ImageIcon(filename);

        // try to read from URL
        if ((icon == null) || (icon.getImageLoadStatus() !=
           MediaTracker.COMPLETE)) {
            try {
                URL url = new URL(filename);
                icon = new ImageIcon(url);
            } catch (MalformedURLException e) {
                /* not a url */
            }
        }

        // in case file is inside a .jar (classpath relative to
           StdDraw)
        /*
```

```
     * if ((icon == null) || (icon.getImageLoadStatus() !=
        MediaTracker.COMPLETE)) {
     * URL url = StdDraw.class.getResource(filename); if (url !=
        null) icon = new
     * ImageIcon(url); }
     */

    // in case file is inside a .jar (classpath relative to root
       of jar)
    if ((icon == null) || (icon.getImageLoadStatus() !=
       MediaTracker.COMPLETE)) {
        URL url = Draw.class.getResource("/" + filename);
        if (url == null)
            throw new IllegalArgumentException("image " +
               filename + " not found");
        icon = new ImageIcon(url);
    }

    return icon.getImage();
}

/**
 * Draws the specified image centered at (<em>x</em>,
    <em>y</em>). The supported
 * image formats are JPEG, PNG, and GIF. As an optimization, the
    picture is
 * cached, so there is no performance penalty for redrawing the
    same image
 * multiple times (e.g., in an animation). However, if you
    change the picture
 * file after drawing it, subsequent calls will draw the
    original picture.
 *
 * @param x         the center <em>x</em>-coordinate of the image
 * @param y         the center <em>y</em>-coordinate of the image
 * @param filename the name of the image/picture, e.g.,
    "ball.gif"
 * @throws IllegalArgumentException if the image filename is
    invalid
 * @throws IllegalArgumentException if either {@code x} or
    {@code y} is either
 *                                  NaN or infinite
 */
public void picture(double x, double y, String filename) {
    validate(x, "x");
    validate(y, "y");
    validateNotNull(filename, "filename");

    Image image = getImage(filename);
```

```
        double xs = scaleX(x);
        double ys = scaleY(y);
        int ws = image.getWidth(null);
        int hs = image.getHeight(null);
        if (ws < 0 || hs < 0)
            throw new IllegalArgumentException("image␣" + filename +
               "␣is␣corrupt");

        offscreen.drawImage(image, (int) Math.round(xs - ws / 2.0),
           (int) Math.round(ys - hs / 2.0), null);
        draw();
    }

    /**
     * Draws the specified image centered at (<em>x</em>,
        <em>y</em>), rotated given
     * number of degrees. The supported image formats are JPEG, PNG,
        and GIF.
     *
     * @param x         the center <em>x</em>-coordinate of the image
     * @param y         the center <em>y</em>-coordinate of the image
     * @param filename the name of the image/picture, e.g.,
        "ball.gif"
     * @param degrees  is the number of degrees to rotate
        counterclockwise
     * @throws IllegalArgumentException if the image filename is
        invalid
     * @throws IllegalArgumentException if {@code x}, {@code y},
        {@code degrees} is
     *                                  NaN or infinite
     * @throws IllegalArgumentException if {@code filename} is
        {@code null}
     */
    public void picture(double x, double y, String filename, double
       degrees) {
        validate(x, "x");
        validate(y, "y");
        validate(degrees, "degrees");
        validateNotNull(filename, "filename");

        Image image = getImage(filename);
        double xs = scaleX(x);
        double ys = scaleY(y);
        int ws = image.getWidth(null);
        int hs = image.getHeight(null);
        if (ws < 0 || hs < 0)
            throw new IllegalArgumentException("image␣" + filename +
               "␣is␣corrupt");
```

```
        offscreen.rotate(Math.toRadians(-degrees), xs, ys);
        offscreen.drawImage(image, (int) Math.round(xs - ws / 2.0),
           (int) Math.round(ys - hs / 2.0), null);
        offscreen.rotate(Math.toRadians(+degrees), xs, ys);

        draw();
    }

    /**
     * Draws the specified image centered at (<em>x</em>,
        <em>y</em>), rescaled to
     * the specified bounding box. The supported image formats are
        JPEG, PNG, and
     * GIF.
     *
     * @param x             the center <em>x</em>-coordinate of the
        image
     * @param y             the center <em>y</em>-coordinate of the
        image
     * @param filename      the name of the image/picture, e.g.,
        "ball.gif"
     * @param scaledWidth   the width of the scaled image (in screen
        coordinates)
     * @param scaledHeight the height of the scaled image (in screen
        coordinates)
     * @throws IllegalArgumentException if either {@code
        scaledWidth} or
     *                                  {@code scaledHeight} is
        negative
     * @throws IllegalArgumentException if the image filename is
        invalid
     * @throws IllegalArgumentException if {@code x} or {@code y} is
        either NaN or
     *                                  infinite
     * @throws IllegalArgumentException if {@code filename} is
        {@code null}
     */
    public void picture(double x, double y, String filename, double
       scaledWidth, double scaledHeight) {
        validate(x, "x");
        validate(y, "y");
        validate(scaledWidth, "scaled width");
        validate(scaledHeight, "scaled height");
        validateNotNull(filename, "filename");
        validateNonnegative(scaledWidth, "scaled width");
        validateNonnegative(scaledHeight, "scaled height");

        Image image = getImage(filename);
        double xs = scaleX(x);
```

```
        double ys = scaleY(y);
        double ws = factorX(scaledWidth);
        double hs = factorY(scaledHeight);
        if (ws < 0 || hs < 0)
            throw new IllegalArgumentException("image␣" + filename +
               "␣is␣corrupt");
        if (ws <= 1 && hs <= 1)
            pixel(x, y);
        else {
            offscreen.drawImage(image, (int) Math.round(xs - ws /
               2.0), (int) Math.round(ys - hs / 2.0),
                    (int) Math.round(ws), (int) Math.round(hs),
                       null);
        }
        draw();
    }

    /**
     * Draws the specified image centered at (<em>x</em>,
        <em>y</em>), rotated given
     * number of degrees, and rescaled to the specified bounding
        box. The supported
     * image formats are JPEG, PNG, and GIF.
     *
     * @param x             the center <em>x</em>-coordinate of the
        image
     * @param y             the center <em>y</em>-coordinate of the
        image
     * @param filename      the name of the image/picture, e.g.,
        "ball.gif"
     * @param scaledWidth   the width of the scaled image (in screen
        coordinates)
     * @param scaledHeight the height of the scaled image (in screen
        coordinates)
     * @param degrees       is the number of degrees to rotate
        counterclockwise
     * @throws IllegalArgumentException if either {@code
        scaledWidth} or
     *                                  {@code scaledHeight} is
        negative
     * @throws IllegalArgumentException if the image filename is
        invalid
     */
    public void picture(double x, double y, String filename, double
       scaledWidth, double scaledHeight, double degrees) {
        validate(x, "x");
        validate(y, "y");
        validate(scaledWidth, "scaled␣width");
        validate(scaledHeight, "scaled␣height");
```

```
        validate(degrees, "degrees");
        validateNotNull(filename, "filename");
        validateNonnegative(scaledWidth, "scaled width");
        validateNonnegative(scaledHeight, "scaled height");

        Image image = getImage(filename);
        double xs = scaleX(x);
        double ys = scaleY(y);
        double ws = factorX(scaledWidth);
        double hs = factorY(scaledHeight);
        if (ws < 0 || hs < 0)
            throw new IllegalArgumentException("image " + filename +
               " is corrupt");
        if (ws <= 1 && hs <= 1)
            pixel(x, y);

        offscreen.rotate(Math.toRadians(-degrees), xs, ys);
        offscreen.drawImage(image, (int) Math.round(xs - ws / 2.0),
           (int) Math.round(ys - hs / 2.0),
                (int) Math.round(ws), (int) Math.round(hs), null);
        offscreen.rotate(Math.toRadians(+degrees), xs, ys);

        draw();
    }

    /***************************************************************************
     * Drawing text.
     ***************************************************************************/

    /**
     * Writes the given text string in the current font, centered at
        (<em>x</em>,
     * <em>y</em>).
     *
     * @param x    the center <em>x</em>-coordinate of the text
     * @param y    the center <em>y</em>-coordinate of the text
     * @param text the text to write
     * @throws IllegalArgumentException if {@code text} is {@code
        null}
     * @throws IllegalArgumentException if {@code x} or {@code y} is
        either NaN or
     *                                  infinite
     */
    public void text(double x, double y, String text) {
        validate(x, "x");
        validate(y, "y");
        validateNotNull(text, "text");

        offscreen.setFont(font);
```

```
        FontMetrics metrics = offscreen.getFontMetrics();
        double xs = scaleX(x);
        double ys = scaleY(y);
        int ws = metrics.stringWidth(text);
        int hs = metrics.getDescent();
        offscreen.drawString(text, (float) (xs - ws / 2.0), (float)
           (ys + hs));
        draw();
    }

    /**
     * Writes the given text string in the current font, centered at
        (<em>x</em>,
     * <em>y</em>) and rotated by the specified number of degrees.
     *
     * @param x        the center <em>x</em>-coordinate of the text
     * @param y        the center <em>y</em>-coordinate of the text
     * @param text     the text to write
     * @param degrees is the number of degrees to rotate
        counterclockwise
     * @throws IllegalArgumentException if {@code text} is {@code
        null}
     * @throws IllegalArgumentException if {@code x}, {@code y}, or
        {@code degrees}
     *                                  is either NaN or infinite
     */
    public void text(double x, double y, String text, double
       degrees) {
        validate(x, "x");
        validate(y, "y");
        validate(degrees, "degrees");
        validateNotNull(text, "text");

        double xs = scaleX(x);
        double ys = scaleY(y);
        offscreen.rotate(Math.toRadians(-degrees), xs, ys);
        text(x, y, text);
        offscreen.rotate(Math.toRadians(+degrees), xs, ys);
    }

    /**
     * Writes the given text string in the current font,
        left-aligned at
     * (<em>x</em>, <em>y</em>).
     *
     * @param x     the <em>x</em>-coordinate of the text
     * @param y     the <em>y</em>-coordinate of the text
     * @param text the text
     * @throws IllegalArgumentException if {@code text} is {@code
```

```
     null}
     * @throws IllegalArgumentException if {@code x} or {@code y} is
        either NaN or
     *                                  infinite
     */
    public void textLeft(double x, double y, String text) {
        validate(x, "x");
        validate(y, "y");
        validateNotNull(text, "text");

        offscreen.setFont(font);
        FontMetrics metrics = offscreen.getFontMetrics();
        double xs = scaleX(x);
        double ys = scaleY(y);
        // int ws = metrics.stringWidth(text);
        int hs = metrics.getDescent();
        offscreen.drawString(text, (float) xs, (float) (ys + hs));
        draw();
    }

    /**
     * Writes the given text string in the current font,
        right-aligned at
     * (<em>x</em>, <em>y</em>).
     *
     * @param x    the <em>x</em>-coordinate of the text
     * @param y    the <em>y</em>-coordinate of the text
     * @param text the text to write
     * @throws IllegalArgumentException if {@code text} is {@code
        null}
     * @throws IllegalArgumentException if {@code x} or {@code y} is
        either NaN or
     *                                  infinite
     */
    public void textRight(double x, double y, String text) {
        validate(x, "x");
        validate(y, "y");
        validateNotNull(text, "text");

        offscreen.setFont(font);
        FontMetrics metrics = offscreen.getFontMetrics();
        double xs = scaleX(x);
        double ys = scaleY(y);
        int ws = metrics.stringWidth(text);
        int hs = metrics.getDescent();
        offscreen.drawString(text, (float) (xs - ws), (float) (ys +
           hs));
        draw();
    }
```

```
/**
 * Copies the offscreen buffer to the onscreen buffer, pauses
    for t milliseconds
 * and enables double buffering.
 *
 * @param t number of milliseconds
 * @deprecated replaced by {@link #enableDoubleBuffering()},
    {@link #show()},
 *             and {@link #pause(int t)}
 */
@Deprecated
public void show(int t) {
    show();
    pause(t);
    enableDoubleBuffering();
}

/**
 * Pause for t milliseconds. This method is intended to support
    computer
 * animations.
 *
 * @param t number of milliseconds
 */
public void pause(int t) {
    try {
        Thread.sleep(t);
    } catch (InterruptedException e) {
        System.out.println("Error sleeping");
    }
}

/**
 * Copies offscreen buffer to onscreen buffer. There is no
    reason to call this
 * method unless double buffering is enabled.
 */
public void show() {
    onscreen.drawImage(offscreenImage, 0, 0, null);
    frame.repaint();
}

// draw onscreen if defer is false
private void draw() {
    if (!defer)
        show();
}
```

```
/**
 * Enable double buffering. All subsequent calls to drawing
    methods such as
 * {@code line()}, {@code circle()}, and {@code square()} will
    be deferred until
 * the next call to show(). Useful for animations.
 */
public void enableDoubleBuffering() {
    defer = true;
}

/**
 * Disable double buffering. All subsequent calls to drawing
    methods such as
 * {@code line()}, {@code circle()}, and {@code square()} will
    be displayed on
 * screen when called. This is the default.
 */
public void disableDoubleBuffering() {
    defer = false;
}

/**
 * Saves the drawing to using the specified filename. The
    supported image
 * formats are JPEG and PNG; the filename suffix must be {@code
    .jpg} or
 * {@code .png}.
 *
 * @param filename the name of the file with one of the required
    suffixes
 * @throws IllegalArgumentException if {@code filename} is
    {@code null}
 */
public void save(String filename) {
    validateNotNull(filename, "filename");
    File file = new File(filename);
    String suffix = filename.substring(filename.lastIndexOf('.')
       + 1);

    // png files
    if ("png".equalsIgnoreCase(suffix)) {
        try {
            ImageIO.write(offscreenImage, suffix, file);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
```

```
        // need to change from ARGB to RGB for jpeg
        // reference:
        //
            http://archives.java.sun.com/cgi-bin/wa?A2=ind0404&L=java2d-interest&
        else if ("jpg".equalsIgnoreCase(suffix)) {
            WritableRaster raster = offscreenImage.getRaster();
            WritableRaster newRaster;
            newRaster = raster.createWritableChild(0, 0, width,
               height, 0, 0, new int[] { 0, 1, 2 });
            DirectColorModel cm = (DirectColorModel)
               offscreenImage.getColorModel();
            DirectColorModel newCM = new
               DirectColorModel(cm.getPixelSize(), cm.getRedMask(),
               cm.getGreenMask(),
                     cm.getBlueMask());
            BufferedImage rgbBuffer = new BufferedImage(newCM,
               newRaster, false, null);
            try {
                ImageIO.write(rgbBuffer, suffix, file);
            } catch (IOException e) {
                e.printStackTrace();
            }
        }

        else {
            System.out.println("Invalid image file type: " + suffix);
        }
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void actionPerformed(ActionEvent e) {
        FileDialog chooser = new FileDialog(frame, "Use a .png or
           .jpg extension", FileDialog.SAVE);
        chooser.setVisible(true);
        String filename = chooser.getFile();
        if (filename != null) {
            save(chooser.getDirectory() + File.separator +
               chooser.getFile());
        }
    }

    /* ********************************************************************
     * Event-based interactions.
     ********************************************************************

    /**
```

```
     * Adds a {@link DrawListener} to listen to keyboard and mouse
       events.
     *
     * @param listener the {\tt DrawListener} argument
     */
    public void addListener(DrawListener listener) {
        // ensure there is a window for listenting to events
        show();
        listeners.add(listener);
        frame.addKeyListener(this);
        frame.addMouseListener(this);
        frame.addMouseMotionListener(this);
        frame.setFocusable(true);
    }

    /***************************************************************************
     * Mouse interactions.
     ***************************************************************************/

    /**
     * Returns true if the mouse is being pressed.
     *
     * @return {@code true} if the mouse is being pressed; {@code
       false} otherwise
     */
    public boolean isMousePressed() {
        synchronized (mouseLock) {
            return isMousePressed;
        }
    }

    /**
     * Returns true if the mouse is being pressed.
     *
     * @return {@code true} if the mouse is being pressed; {@code
       false} otherwise
     * @deprecated replaced by {@link #isMousePressed()}
     */
    @Deprecated
    public boolean mousePressed() {
        synchronized (mouseLock) {
            return isMousePressed;
        }
    }

    /**
     * Returns the x-coordinate of the mouse.
     *
     * @return the x-coordinate of the mouse
```

```
     */
    public double mouseX() {
        synchronized (mouseLock) {
            return mouseX;
        }
    }

    /**
     * Returns the y-coordinate of the mouse.
     *
     * @return the y-coordinate of the mouse
     */
    public double mouseY() {
        synchronized (mouseLock) {
            return mouseY;
        }
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mouseEntered(MouseEvent e) {
        // this body is intentionally left empty
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mouseExited(MouseEvent e) {
        // this body is intentionally left empty
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mousePressed(MouseEvent e) {
        synchronized (mouseLock) {
            mouseX = userX(e.getX());
            mouseY = userY(e.getY());
            isMousePressed = true;
        }
        if (e.getButton() == MouseEvent.BUTTON1) {
            for (DrawListener listener : listeners)
                listener.mousePressed(userX(e.getX()),
                    userY(e.getY()));
        }
```

```
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mouseReleased(MouseEvent e) {
        synchronized (mouseLock) {
            isMousePressed = false;
        }
        if (e.getButton() == MouseEvent.BUTTON1) {
            for (DrawListener listener : listeners)
                listener.mouseReleased(userX(e.getX()),
                    userY(e.getY()));
        }
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mouseClicked(MouseEvent e) {
        if (e.getButton() == MouseEvent.BUTTON1) {
            for (DrawListener listener : listeners)
                listener.mouseClicked(userX(e.getX()),
                    userY(e.getY()));
        }
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mouseDragged(MouseEvent e) {
        synchronized (mouseLock) {
            mouseX = userX(e.getX());
            mouseY = userY(e.getY());
        }
        // doesn't seem to work if a button is specified
        for (DrawListener listener : listeners)
            listener.mouseDragged(userX(e.getX()), userY(e.getY()));
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void mouseMoved(MouseEvent e) {
```

```
        synchronized (mouseLock) {
            mouseX = userX(e.getX());
            mouseY = userY(e.getY());
        }
    }

    /* ***************************************************************************
     * Keyboard interactions.
     *************************************************************************** */

    /**
     * Returns true if the user has typed a key.
     *
     * @return {@code true} if the user has typed a key; {@code
        false} otherwise
     */
    public boolean hasNextKeyTyped() {
        synchronized (keyLock) {
            return !keysTyped.isEmpty();
        }
    }

    /**
     * The next key typed by the user.
     *
     * @return the next key typed by the user
     */
    public char nextKeyTyped() {
        synchronized (keyLock) {
            return keysTyped.removeLast();
        }
    }

    /**
     * Returns true if the keycode is being pressed.
     * <p>
     * This method takes as an argument the keycode (corresponding
        to a physical
     * key). It can handle action keys (such as F1 and arrow keys)
        and modifier keys
     * (such as shift and control). See {@link KeyEvent} for a
        description of key
     * codes.
     *
     * @param keycode the keycode to check
     * @return {@code true} if {@code keycode} is currently being
        pressed;
     *         {@code false} otherwise
     */
```

```
    public boolean isKeyPressed(int keycode) {
        synchronized (keyLock) {
            return keysDown.contains(keycode);
        }
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void keyTyped(KeyEvent e) {
        synchronized (keyLock) {
            keysTyped.addFirst(e.getKeyChar());
        }

        // notify all listeners
        for (DrawListener listener : listeners)
            listener.keyTyped(e.getKeyChar());
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void keyPressed(KeyEvent e) {
        synchronized (keyLock) {
            keysDown.add(e.getKeyCode());
        }

        // notify all listeners
        for (DrawListener listener : listeners)
            listener.keyPressed(e.getKeyCode());
    }

    /**
     * This method cannot be called directly.
     */
    @Override
    public void keyReleased(KeyEvent e) {
        synchronized (keyLock) {
            keysDown.remove(e.getKeyCode());
        }

        // notify all listeners
        for (DrawListener listener : listeners)
            listener.keyReleased(e.getKeyCode());
    }

    /*****************************************************************************
```

```
     * For improved resolution on Mac Retina displays.
     *****************************************************************************/

    @SuppressWarnings("serial")
    private static class RetinaImageIcon extends ImageIcon {

        public RetinaImageIcon(Image image) {
            super(image);
        }

        public int getIconWidth() {
            return super.getIconWidth() / 2;
        }

        /**
         * Gets the height of the icon.
         *
         * @return the height in pixels of this icon
         */
        public int getIconHeight() {
            return super.getIconHeight() / 2;
        }

        public synchronized void paintIcon(Component c, Graphics g,
           int x, int y) {
            Graphics2D g2 = (Graphics2D) g.create();
            g2.setRenderingHint(RenderingHints.KEY_INTERPOLATION,
                RenderingHints.VALUE_INTERPOLATION_BICUBIC);
            g2.setRenderingHint(RenderingHints.KEY_RENDERING,
                RenderingHints.VALUE_RENDER_QUALITY);
            g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
                RenderingHints.VALUE_ANTIALIAS_ON);
            g2.scale(0.5, 0.5);
            super.paintIcon(c, g2, x * 2, y * 2);
            g2.dispose();
        }
    }

    /**
     * Test client.
     *
     * @param args the command-line arguments
     */
    public static void main(String[] args) {

        // create one drawing window
        Draw draw1 = new Draw("Test client 1");
        draw1.square(0.2, 0.8, 0.1);
        draw1.filledSquare(0.8, 0.8, 0.2);
```

```
        draw1.circle(0.8, 0.2, 0.2);
        draw1.setPenColor(Draw.MAGENTA);
        draw1.setPenRadius(0.02);
        draw1.arc(0.8, 0.2, 0.1, 200, 45);

        // create another one
        Draw draw2 = new Draw("Test client 2");
        draw2.setCanvasSize(900, 200);
        // draw a blue diamond
        draw2.setPenRadius();
        draw2.setPenColor(Draw.BLUE);
        double[] x = { 0.1, 0.2, 0.3, 0.2 };
        double[] y = { 0.2, 0.3, 0.2, 0.1 };
        draw2.filledPolygon(x, y);

        // text
        draw2.setPenColor(Draw.BLACK);
        draw2.text(0.2, 0.5, "bdfdfdfdlack text");
        draw2.setPenColor(Draw.WHITE);
        draw2.text(0.8, 0.8, "white text");
    }

}
```

```
package graphics;

/*  Compilation:  javac DrawListener.java
 *  Execution:    none
 *  Dependencies: none
 *
 *  Interface that accompanies Draw.java.
 *************************************************************************/

/**
 * <i>DrawListener</i>. This interface provides a basic capability
    for
 * responding to keyboard in mouse events from {@link Draw} via
    callbacks. You
 * can see some examples in
 * <a
    href="https://introcs.cs.princeton.edu/java/36inheritance">Section
 * 3.6</a>.
 *
 * <p>
 * For additional documentation, see
 * <a href="https://introcs.cs.princeton.edu/31datatype">Section
    3.1</a> of
 * <i>Computer Science: An Interdisciplinary Approach</i> by Robert
    Sedgewick
```

```
 * and Kevin Wayne.
 *
 * @author Robert Sedgewick
 * @author Kevin Wayne
 */

public interface DrawListener {

    /**
     * Invoked when the mouse has been pressed.
     *
     * @param x the x-coordinate of the mouse
     * @param y the y-coordinate of the mouse
     */
    void mousePressed(double x, double y);

    /**
     * Invoked when the mouse has been dragged.
     *
     * @param x the x-coordinate of the mouse
     * @param y the y-coordinate of the mouse
     */
    void mouseDragged(double x, double y);

    /**
     * Invoked when the mouse has been released.
     *
     * @param x the x-coordinate of the mouse
     * @param y the y-coordinate of the mouse
     */
    void mouseReleased(double x, double y);

    /**
     * Invoked when the mouse has been clicked (pressed and
       released).
     *
     * @param x the x-coordinate of the mouse
     * @param y the y-coordinate of the mouse
     */
    void mouseClicked(double x, double y);

    /**
     * Invoked when a key has been typed.
     *
     * @param c the character typed
     */
    void keyTyped(char c);

    /**
```

```
     * Invoked when a key has been pressed.
     *
     * @param keycode the key combination pressed
     */
    void keyPressed(int keycode);

    /**
     * Invoked when a key has been released.
     *
     * @param keycode the key combination released
     */
    void keyReleased(int keycode);
}
```

```
package graphics;

import java.awt.Dimension;

import javax.swing.JFrame;

/**
 * Objet graphique cadre pour LocalGraphics
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public class Frame extends JFrame {

    private static final long serialVersionUID = 1L;

    /**
     * La fenetre contenue par le cadre
     */
    public Window window;

    /**
     * Constructeur simple
     *
     * @param width       : la largeur en pixels de la fenetre
     * @param height      : la hauteur en pixels de la fenetre
     * @param localGraphics : l'objet graphique (galement listener)
     */
    public Frame(int width, int height, LocalGraphics localGraphics)
        {
         this.setTitle("Simulation");
         this.window = new Window(width, height);
         this.setSize(new Dimension(width, height));
         this.getContentPane().add(window);
```

```
        this.addKeyListener(localGraphics);
        this.addMouseListener(localGraphics);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setResizable(false);
        this.setLocationRelativeTo(null);
        this.setLayout(null);
        this.setVisible(true);
    }
}
```

```
package graphics;

import java.awt.Color;
import java.util.LinkedList;

import algorithms.ClusterAttribution;
import structure.Checkpoint;
import structure.Cluster;
import structure.Drone;
import structure.Vector;

/**
 * Classe contenant les fonctions d'affichage
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public class Graphics implements GraphicsInterface {

    /**
     * Objet graphique
     */
    private final Draw draw;

    /**
     * Constructeur simple sur 2000 * 2000 pixels
     */
    public Graphics(int width, int height) {
        draw = new Draw();
        draw.setCanvasSize(width, height);
    }

    /**
     * Met    jour les graphique pour une configuration
     *
     * @param clusterAttribution : la configuration
     */
    public void updateGraphics(ClusterAttribution
```

```
    clusterAttribution) {
        draw.setPenColor(Draw.WHITE);
        draw.filledRectangle(0, 0, 1, 1);
        LinkedList<Drone> drones = clusterAttribution.getDrones();
        int numberOfDrones = drones.size();
        int i = 0;
        for (Drone drone : drones) {
            Color color = Color.getHSBColor(((float) i) / ((float)
                numberOfDrones), 1f, 1f);
            draw.setPenColor(color);
            updateGraphicsForDrone(drone,
                clusterAttribution.getDroneCluster(drone));
            i++;
        }
    }

    /**
     * Dessine un drone et son chemin
     *
     * @param drone   : le drone
     * @param cluster : le cluster du drone
     */
    private void updateGraphicsForDrone(Drone drone, Cluster
        cluster) {
        paintDrone(drone);
        paintDronePath(cluster);
    }

    /**
     * Dessine un drone
     *
     * @param drone : le drone
     */
    private void paintDrone(Drone drone) {
        draw.setPenRadius(0.01);
        draw.square(drone.getX(), drone.getY(), 0.01);
    }

    /**
     * Dessine un cluster
     *
     * @param cluster : le cluster
     */
    private void paintDronePath(Cluster cluster) {
        draw.setPenRadius(0.01);
        LinkedList<Checkpoint> checkpoints =
            cluster.getCheckpointsOrdered();
        if (checkpoints.size() == 0)
            return;
```

```
        if (checkpoints.size() == 1) {
            paintCheckpoint(checkpoints.getFirst());
            return;
        }
        for (int i = 0; i < checkpoints.size() - 1; i++) {
            paintCheckpoint(checkpoints.get(i));
            paintLine(checkpoints.get(i), checkpoints.get(i + 1));
        }
        paintCheckpoint(checkpoints.getLast());
        paintLine(checkpoints.getLast(), checkpoints.getFirst());
    }

    /**
     * Fonction auxiliaire pour dessiner une ligne
     *
     * @param firstPosition  : la premi re extr mit  de la ligne
     * @param secondPosition : la seconde extr mit  de la ligne
     */

    private void paintLine(Vector firstPosition, Vector
       secondPosition) {
        draw.line(firstPosition.getX(), firstPosition.getY(),
           secondPosition.getX(), secondPosition.getY());
    }

    /**
     * Dessine un checkpoint
     *
     * @param checkpoint
     */
    private void paintCheckpoint(Checkpoint checkpoint) {
        draw.setPenRadius(0.02);
        draw.point(checkpoint.getX(), checkpoint.getY());
    }
}
```

```
package graphics;

import algorithms.ClusterAttribution;

/**
 * Interface pour les objets graphiques
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public interface GraphicsInterface {
```

```
    /**
     * Dessine une configuration
     *
     * @param configuration : la configuration
     */
    public void updateGraphics(ClusterAttribution configuration);

}
```

```
package graphics;

import java.awt.Graphics;
import java.awt.Point;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;
import java.util.LinkedList;

import algorithms.ClusterAttribution;
import algorithms.ImproveType;
import structure.Checkpoint;
import structure.Cluster;
import structure.Drone;
import structure.Vector;

/**
 * Objet graphique pour un affichage personnalis  et plus rapide
    que Draw, @see
 * {@link Draw}
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public class LocalGraphics implements MouseListener, KeyListener,
   GraphicsInterface {

    /**
     * Le cadre
     */
    private final Frame frame;

    /**
     * La fen tre
     */
    private final Window window;

    /**
```

```
     * L'objet graphique de la fen tre
     */
    private final Graphics graphics;

    /**
     * L' tat  de la souris. Conditionne l'action  effectuer en
        cas de clic
     */
    public MouseState mouseState;

    /**
     * La largeur de la fen tre
     */
    private final int width;

    /**
     * La hauteur de la fen tre
     */
    private final int height;

    /**
     * Constructeur simple. Par d faut, la souris est sur le mode
        EMPTY
     *
     * @param width  : la largeur de la fen tre
     * @param height : la hauteur de la fen tre
     */
    public LocalGraphics(int width, int height) {
        this.width = width;
        this.height = height;
        this.frame = new Frame(width, height, this);
        this.window = frame.window;
        this.graphics = window.getGraphics();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            System.out.println("Interruption impromptue");
        }
        mouseState = MouseState.EMPTY;
    }

    @Override
    public void updateGraphics(ClusterAttribution
       clusterAttribution) {
        window.reset(graphics);
        LinkedList<Drone> drones = clusterAttribution.getDrones();
        int numberOfDrones = drones.size();
        int i = 0;
        for (Drone drone : drones) {
```

```
            window.setColor(i, numberOfDrones, graphics);
            updateGraphicsForDrone(drone,
                clusterAttribution.getDroneCluster(drone));
            i++;
        }
    }

    /**
     * Met    jour les graphiques pour un drone et son cluster
     *
     * @param drone   : le drone
     * @param cluster : le cluster
     */
    private void updateGraphicsForDrone(Drone drone, Cluster
        cluster) {
        window.paintDrone(drone, graphics);
        window.paintCluster(cluster, graphics);
    }

    @Override
    public void keyTyped(KeyEvent e) {

    }

    @Override
    public void keyPressed(KeyEvent e) {
        if (e.getKeyChar() == 'c')
            mouseState = MouseState.CHECKPOINT;
        else if (e.getKeyChar() == 'd')
            mouseState = MouseState.DRONE;
        else if (e.getKeyChar() == 'e')
            mouseState = MouseState.EMPTY;

    }

    @Override
    public void keyReleased(KeyEvent e) {
    }

    @Override
    public void mouseClicked(MouseEvent e) {
        Point point = e.getPoint();
        Vector vector = new Vector(point.getX() / width,
            point.getY() / height);
        if (mouseState == MouseState.CHECKPOINT) {
            Controller.addCheckpoint(new Checkpoint(vector),
                ImproveType.COMPLETE);
        } else if (mouseState == MouseState.DRONE) {
            Controller.addDrone(new Drone(vector),
```

```
                    ImproveType.COMPLETE);
            }
        }

        @Override
        public void mousePressed(MouseEvent e) {
        }

        @Override
        public void mouseReleased(MouseEvent e) {

        }

        @Override
        public void mouseEntered(MouseEvent e) {

        }

        @Override
        public void mouseExited(MouseEvent e) {

        }

        /**
         * Enumration des actions possibles avec la souris
         *
         * @LouisProffitX
         * @author Louis Proffit
         * @version 1.0
         */
        private static enum MouseState {
            /**
             * Supprimer
             */
            EMPTY,

            /**
             * Ajouter un checkpoint
             */
            CHECKPOINT,

            /**
             * Ajouter un drone
             */
            DRONE
        }
}
```

```
package graphics;
```

```java
import java.awt.Color;
import java.awt.Graphics;
import java.util.LinkedList;

import javax.swing.JPanel;

import structure.Checkpoint;
import structure.Cluster;
import structure.Drone;
import structure.Vector;

/**
 * Fen tre graphique pour un {@link LocalGraphics}
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public class Window extends JPanel {

    private static final long serialVersionUID = 1L;

    public static final double droneSize = 0.015;
    public static final double checkpointSize = 0.01;

    /**
     * La largeur
     */
    private final int width;

    /**
     * La hauteur
     */
    private final int height;

    /**
     * 95% de la largeur, pour garder une marge
     */
    private final int resizedWidth;

    /**
     * 95% de la hauteur, pour garder une marge
     */
    private final int resizedHeight;

    /**
     * Constructeur simple
     *
```

```
     * @param width  : la largeur de la fen tre
     * @param height : la hauteur de la fen tre
     */
    public Window(int width, int height) {
        super();
        this.setOpaque(false);
        this.setBounds(0, 0, width, height);
        this.setBackground(Color.WHITE);
        this.width = width;
        this.height = height;
        this.resizedWidth = (int) (width * 0.95);
        this.resizedHeight = (int) (width * 0.95);
    }

    /**
     * Configure une couleur dans une palette
     *
     * @param index    : l'indice de la couleur dans la palette
     * @param size     : la taille de la palette
     * @param graphics : l'objet auquel appliquer la nouvelle couleur
     */
    public void setColor(int index, int size, Graphics graphics) {
        graphics.setColor(Color.getHSBColor(((float) index) /
            ((float) size), 1f, 1f));
    }

    /**
     * Remet    jour le graphisme de la fen tre
     *
     * @param g
     */
    public void reset(Graphics g) {
        g.setColor(Color.WHITE);
        g.fillRect(0, 0, width, height);
    }

    /**
     * Dessine un drone
     *
     * @param drone : le drone
     */
    public void paintDrone(Drone drone, Graphics g) {
        int size = getRoundedX(droneSize);
        int x = getRoundedX(drone.getX() - droneSize / 2);
        int y = getRoundedX(drone.getY() - droneSize / 2);
        g.fillOval(x, y, size, size);
    }

    /**
```

```
     * Dessine un checkpoint
     *
     * @param checkpoint : le checkpoint
     */
    private void paintCheckpoint(Checkpoint checkpoint, Graphics g) {
        int size = getRoundedX(checkpointSize);
        int x = getRoundedX(checkpoint.getX() - checkpointSize / 2);
        int y = getRoundedX(checkpoint.getY() - checkpointSize / 2);
        g.fillOval(x, y, size, size);
    }

    /**
     * Dessine un cluster
     *
     * @param cluster : le cluster
     */
    public void paintCluster(Cluster cluster, Graphics g) {
        LinkedList<Checkpoint> checkpoints =
            cluster.getCheckpointsOrdered();
        if (checkpoints.size() == 0)
            return;
        if (checkpoints.size() == 1) {
            paintCheckpoint(checkpoints.getFirst(), g);
            return;
        }
        for (int i = 0; i < checkpoints.size() - 1; i++) {
            paintCheckpoint(checkpoints.get(i), g);
            paintLine(checkpoints.get(i), checkpoints.get(i + 1), g);
        }
        paintCheckpoint(checkpoints.getLast(), g);
        paintLine(checkpoints.getLast(), checkpoints.getFirst(), g);
    }

    /**
     * Fonction auxiliaire pour dessiner une ligne
     *
     * @param firstPosition  : la premi re extr mit  de la ligne
     * @param secondPosition : la seconde extr mit  de la ligne
     */
    private void paintLine(Vector firstPosition, Vector
        secondPosition, Graphics g) {
        g.drawLine(getRoundedX(firstPosition.getX()),
            getRoundedX(firstPosition.getY()),
                getRoundedX(secondPosition.getX()),
                    getRoundedX(secondPosition.getY()));
    }

    /** R cup re un x resized en pixel  partir d'un x dans le
        cadre */
```

```
    public int getRoundedX(double x) {
        return (int) (x * resizedWidth);
    }

    /** R cup re un y resized en pixel    partir d'un y dans le
       cadre */
    public int getRoundedY(double y) {
        return (int) (y * resizedHeight);
    }
}
```

1.3 Package structure

```
package structure;

/**
 * Classe repr sentant un checkpoint
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public class Checkpoint extends Vector {

    /**
     * Constructeur simple
     *
     * @param position
     */
    public Checkpoint(Vector position) {
        super(position.getX(), position.getY());
    }
}
```

```
package structure;

import java.util.LinkedList;

import algorithms.RecuitInterface;
import algorithms.TSPRecuit;

/**
 * Classe repr sentant un {@link Cluster}. Contient la possibilit
    d' tre
 * am lior e par un algorithme {@link TSPRecuit}.
 *
 * @LouisProffitX
```

```
 * @author Louis Proffit
 * @version 1.0
 */
public class Cluster implements RecuitInterface {

    /**
     * La cible en cours dans le cluster
     */
    private Checkpoint currentTarget = null;

    /**
     * Le chemin du cluster
     */
    private Path path = new Path();

    /**
     * Effectue un passage de {@link TSPRecuit} pour am liorer le
        chemin
     */
    public void improvePath() {
        TSPRecuit.improvePath(this);
    }

    /**
     * Renvoie la distance entre le cluster (sa moyenne) et une
        position
     *
     * @param vector : la position
     * @return la distance
     */
    public double distance(Vector vector) {
        return path.distance(vector);
    }

    /**
     * Renvoie la cible courante
     *
     * @return : la cible courante
     */
    public Checkpoint getCurrentTarget() {
        return currentTarget;
    }

    /**
     * Avance d'un cran la cible. Si il n'y a pas de successeur, la
        cible devient
     * nulle
     */
    public void moveTargetForward() {
```

```
        if (currentTarget == null)
            return;
        this.currentTarget = path.getCheckpointAfter(currentTarget);
    }

    /**
     * Renvoie la liste des checkpoints du chemin dans l'ordre de
        parcours (le
     * premier    lment    est arbitraire)
     *
     * @return : la liste ordonn e des checkpoints
     */
    public LinkedList<Checkpoint> getCheckpointsOrdered() {
        return path.getCheckpointsOrdered();
    }

    /**
     * Ajoute un checkpoint au chemin
     *
     * @param checkpoint : le checkpoint    ajouter
     */
    public void addCheckpoint(Checkpoint checkpoint) {
        if (currentTarget == null)
            currentTarget = checkpoint;
        path.addCheckpoint(checkpoint);
    }

    /**
     * Retire un checkpoint du chemin. Si ce checkpoint est la
        cible, on passe    la
     * suivante
     *
     * @param checkpoint : le checkpoint    retirer
     */
    public void removeCheckpoint(Checkpoint checkpoint) {
        if (currentTarget == checkpoint)
            currentTarget = path.getCheckpointAfter(checkpoint);
        path.removeCheckpoint(checkpoint);
    }

    public void clear() {
        currentTarget = null;
        path.clear();
    }

    @Override
    public int getSize() {
        return path.getSize();
    }
```

```
@Override
public Modification modificationFunction() {
    int size = path.getSize();
    return new Pair<Integer>((int) (Math.random() * size), (int)
       (Math.random() * size));
}

@Override
@SuppressWarnings("unchecked")
public Double improvementFunction(Modification modification) {
    double result = 0;
    Pair<Integer> swap = (Pair<Integer>) modification;
    Checkpoint firstCheckpoint =
       path.getCheckpointAtIndex(swap.getFirst());
    Checkpoint secondCheckpoint =
       path.getCheckpointAtIndex(swap.getSecond());
    Checkpoint checkpointBeforeFirst =
       path.getCheckpointBefore(firstCheckpoint);
    Checkpoint checkpointAfterSecond =
       path.getCheckpointAfter(secondCheckpoint);
    result += checkpointBeforeFirst.distance(secondCheckpoint);
    result += checkpointAfterSecond.distance(firstCheckpoint);
    result -= firstCheckpoint.distance(checkpointBeforeFirst);
    result -= secondCheckpoint.distance(checkpointAfterSecond);
    return result;
}

@Override
@SuppressWarnings("unchecked")
public void commitFunction(Modification modification) {
    Pair<Integer> swap = (Pair<Integer>) modification;
    int size = path.getSize();
    int increasingIndex = swap.getFirst();
    int decreasingIndex = swap.getSecond();
    if (increasingIndex == decreasingIndex)
        return;
    if (increasingIndex < decreasingIndex) {
        while (increasingIndex < decreasingIndex) {
            path.swapOrder(increasingIndex, decreasingIndex);
            increasingIndex++;
            decreasingIndex--;
        }
    } else {
        boolean changed = false;
        while (increasingIndex < decreasingIndex & !changed) {
            path.swapOrder(increasingIndex, decreasingIndex);
            increasingIndex++;
            decreasingIndex--;
```

```
                if (increasingIndex == size) {
                    changed = true;
                    increasingIndex = 0;
                }
                if (increasingIndex == -1) {
                    changed = true;
                    decreasingIndex = size - 1;
                }
            }
        }
    }
}
```

```
package structure;

/**
 * Classe repr sentant un {@link Drone}
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public class Drone extends MutableVector {

    /**
     * La vitesse du drone en unit  de cadre par mise    jour
        graphique
     */
    public static double speed = 0.01;

    /**
     * Constructeur simple
     *
     * @param position
     */
    public Drone(Vector position) {
        super(position.getX(), position.getY());
    }

    /**
     * Fait avancer un drone vers une cible sur la distance speed
     *
     * @param target : la cible
     */
    public void move(Vector target) {
        MutableVector movement = new MutableVector(target.getX() -
            this.getX(), target.getY() - this.getY());
        movement.normalize(speed);
        this.add(movement);
```

```
    }
}
```

```
package structure;

/**
 * Interface vide pour un objeet modification, pour le
 * {@link algorithms.TSPRecuit}.
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public interface Modification {

}
```

```
package structure;

/**
 * Classe d crivant un point/vecteur modifiable par ses
    coordonn es cart siennes
 *
 * @author Louis Proffit
 */
public class MutableVector extends Vector {

    /**
     * Constructeur g n rant un point al atoire dans le cadre
     */
    public MutableVector() {
        super(Math.random(), Math.random());
    }

    /**
     * Constructeur simple
     *
     * @param x : L'abscisse du vecteur
     * @param y : L'ordonn e du vecteur
     */
    public MutableVector(double x, double y) {
        super(x, y);
    }

    /**
     * settter
     *
     * @param x : La coordonn e x du vecteur
     */
```

```
    public void setX(double x) {
        this.x = x;
    }

    /**
     * settter
     *
     * @param x : La coordonn e y du vecteur
     */
    public void setY(double y) {
        this.y = y;
    }

    public void set(Vector vector) {
        setX(vector.x);
        setY(vector.y);
    }

    /**
     * M thode pour ajouter un vecteur (avec modification)
     *
     * @param vector : Le vecteur     ajouter
     */
    public void add(Vector vector) {
        this.x += vector.x;
        this.y += vector.y;
    }

    /**
     * M thode pour retirer un vecteur (avec modification)
     *
     * @param vector : Le vecteur     retirer
     */
    public void substract(MutableVector vector) {
        this.x -= vector.x;
        this.y -= vector.y;
    }

    /**
     * Normalise le vecteur
     *
     * @param newNorm : la future norme (non nulle)
     */
    public void normalize(double newNorm) {
        double norm = getNorm();
        this.x = x / norm * newNorm;
        this.y = y / norm * newNorm;
    }
}
```

```java
package structure;

/**
 * Classe d crivant une paire d'objets de m me type
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public class Pair<T> implements Modification {

    /**
     * Le premier    lment    de la paire
     */
    private final T first;

    /**
     * Le second    lment    de la paire
     */
    private final T second;

    /**
     * Constructeur simple
     *
     * @param first  : le premier    lment
     * @param second : le second    lment
     */
    public Pair(T first, T second) {
        this.first = first;
        this.second = second;
    }

    /**
     * Renvoie le premier    lment    de la paire
     *
     * @return : le premier    lment
     */
    public T getFirst() {
        return first;
    }

    /**
     * Renvoie le second    lment    de la paire
     *
     * @return : le second    lment
     */
    public T getSecond() {
        return second;
    }
```

```
    /**
     * Méthode usuelle equals. Deux paires sont égales si elles
        contiennent les mêmes
     *    éléments, indépendament de l'ordre
     */
    @Override
    @SuppressWarnings("unchecked")
    public boolean equals(Object obj) {
        Pair<T> o = (Pair<T>) obj;
        return ((first.equals(o.getFirst()) &
           second.equals(o.getSecond()))
                | (second.equals(o.getFirst()) &
                   first.equals(o.getSecond())));
    }

}
```

```
package structure;

import java.util.HashMap;
import java.util.LinkedList;

/**
 * Classe décrivant un chemin ordonné de {@link Checkpoint}
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public class Path {

    /**
     * Le nombre de {@link Checkpoint} du chemin
     */
    private int size = 0;

    /**
     * La matrice d'association indice -> checkpoint. Contient les
        entrées inverses
     * de indices
     */
    private HashMap<Integer, Checkpoint> order = new HashMap<>();

    /**
     * La matrice d'association checkpoint -> indice. Contient les
        entrées inverses
     * de order
     */
```

```
private HashMap<Checkpoint, Integer> indices = new HashMap<>();

/**
 * Le vecteur moyen des positions des    lments    du chemin
 */
private Vector mean = new Vector();

/**
 * Renvoie la distance entre le chemin (sa moyenne) et une
    position
 *
 * @param vector : la position
 * @return la distance
 */
public double distance(Vector vector) {
    return vector.distance(mean);
}

/**
 * Calcule la moyenne et la stocke dans mean. Si le chemin est
    vide, il attribue
 * un vecteur al atoire
 */
private void computeMean() {
    if (size == 0)
        this.mean = new MutableVector();
    else {
        MutableVector result = new MutableVector(0, 0);
        for (Checkpoint checkpoint : indices.keySet()) {
            result.add(checkpoint);
        }
        this.mean = result.getMult(1f / size);
    }
}

/**
 * Calcule le checkpoint qui suit un autre checkpoint dans le
    chemin
 *
 * @param checkpoint : le checkpoint
 * @return le checkpoint qui suit
 */
public Checkpoint getCheckpointAfter(Checkpoint checkpoint) {
    if (size <= 1)
        return null;
    else {
        int currentIndex = indices.get(checkpoint);
        if (currentIndex < size - 1)
            return order.get(currentIndex + 1);
```

```
            else
                return order.get(0);
        }
    }

    /**
     * Calcule le checkpoint qui précède un autre checkpoint dans
        le chemin
     *
     * @param checkpoint : le checkpoint
     * @return le checkpoint qui précède
     */
    public Checkpoint getCheckpointBefore(Checkpoint checkpoint) {
        if (size == 1)
            return null;
        else {
            int currentIndex = indices.get(checkpoint);
            if (currentIndex > 0)
                return order.get(currentIndex - 1);
            else
                return order.get(size - 1);
        }
    }

    /**
     * Récupère le checkpoint situé à un indice précis
     *
     * @param index : l'indice
     * @return le checkpoint à cet indice
     */
    public Checkpoint getCheckpointAtIndex(int index) {
        return order.get(index);
    }

    /**
     * Récupère l'indice d'un checkpoint précis
     *
     * @param checkpoint : le checkpoint
     * @return l'indice de ce checkpoint
     */
    public int getCheckpointIndex(Checkpoint checkpoint) {
        return indices.get(checkpoint);
    }

    /**
     * Récupère la liste des checkpoints du chemin dans l'ordre
     *
     * @return : la liste des éléments du chemin
     */
```

```
public LinkedList<Checkpoint> getCheckpointsOrdered() {
    LinkedList<Checkpoint> result = new LinkedList<>();
    for (int i = 0; i < size; i++) {
        result.add(order.get(i));
    }
    return result;
}

/**
 * Ajoute un checkpoint dans le chemin (   la derin re position)
 *
 * @param checkpoint : le checkpoint    ajouter
 */
public void addCheckpoint(Checkpoint checkpoint) {
    order.put(size, checkpoint);
    indices.put(checkpoint, size);
    size++;
    computeMean();
}

/**
 * Enl ve un checkpoin du chemin
 *
 * @param checkpoint : le checkpoint    ajouter
 */
public void removeCheckpoint(Checkpoint checkpoint) {
    int index = indices.get(checkpoint);
    swapOrder(index, size - 1);
    removeLastCheckpoint();
}

/**
 * Enl ve le dernier checkpoint du chemin
 */
private void removeLastCheckpoint() {
    Checkpoint previousLastCheckpoint = order.remove(size - 1);
    indices.remove(previousLastCheckpoint);
    size--;
    computeMean();
}

/**
 * Echange les deux indices dans le parcours
 *
 * @param firstIndex  : le premier indice
 * @param secondIndex : le deuxi me indice
 */
public void swapOrder(int firstIndex, int secondIndex) {
    Checkpoint checkpoint1 = order.get(firstIndex);
```

```
        Checkpoint checkpoint2 = order.put(secondIndex, checkpoint1);
        order.put(firstIndex, checkpoint2);
        indices.put(checkpoint1, secondIndex);
        indices.put(checkpoint2, firstIndex);
    }

    /**
     * Renvoie la taille du chemin
     *
     * @return la taille du chemin
     */
    public int getSize() {
        return size;
    }

    public void clear() {
        size = 0;
        order.clear();
        indices.clear();
        mean = new Vector();
    }
}
```

```
package structure;

/**
 * Classe d crivant un vecteur (immodifiable)
 *
 * @LouisProffitX
 * @author Louis Proffit
 * @version 1.0
 */
public class Vector {

    /**
     * La coordonn e x du vecteur
     */
    public double x;

    /**
     * La coordonn e y du vecteur
     */
    public double y;

    /**
     * Constructeur simple d'un point al atoire dans le cadre
     */
    public Vector() {
        this.x = Math.random();
```

```
        this.y = Math.random();
    }

    /**
     * Constructeur simple
     *
     * @param x : l'abscisse du vecteur
     * @param y : l'ordonn e du vecteur
     */
    public Vector(double x, double y) {
        this.x = x;
        this.y = y;
    }

    /**
     * Calcule la distance entre this et un vecteur (vus comme des
        positions)
     *
     * @param vector : l'autre vecteur
     * @return la distance
     */
    public double distance(Vector vector) {
        return Math.pow(Math.pow(vector.x - this.x, 2) +
            Math.pow(vector.y - this.y, 2), 0.5);
    }

    /**
     * Renvoie la coordonn e x du vecteur
     *
     * @return la coordonn e x du vecteur
     */
    public double getX() {
        return x;
    }

    /**
     * Renvoie la coordonn e y du vecteur
     *
     * @return la coordonn e y du vecteur
     */
    public double getY() {
        return y;
    }

    /**
     * Renvoie une copie de this soustrait d'un vecteur vector. this
        n'est pas
     * modifi
     *
```

```
     * @param vector : le vecteur    soustraire
     * @return la copie du vecteur soustrait
     */
    public Vector copyMinus(Vector vector) {
        return new Vector(this.x - vector.x, this.y - vector.y);
    }

    /**
     * Renvoie une copie de this modifi  d'un coefficient mult.
        this n'est pas
     * modifi
     *
     * @param mult : le coefficient
     * @return la copie du vecteur multipli
     */
    public Vector getMult(double mult) {
        return new Vector(this.x * mult, this.y * mult);
    }

    /**
     * Renvoie la norme de this
     *
     * @return la norme de this
     */
    public double getNorm() {
        return Math.pow(Math.pow(this.x, 2) + Math.pow(this.y, 2),
            0.5);
    }

    @Override
    public String toString() {
        return "Vecteur : (" + x + ";" + y + ")";
    }
}
```

2 Méthode par conflit

Le projet Java contient deux packages : le package général et le package graphics qui contient les classes graphiques.

2.1 Package general

```
package general;

public class Checkpoint {

    private Vector position;

    private double satisfaction; // Satisfaction du checkpoint,
        ale   au temps     oul depuis le dernier passage d'un drone

    public Checkpoint(double x, double y) {
        this.position = new Vector(x, y);
        this.satisfaction = 1;
    }

    public Vector getPosition() {
        return position;
    }

    public void setPosition(Vector position) {
        this.position = position;
    }

    public double getSatisfaction() {
        return satisfaction;
    }

    public void setSatisfaction(double satisfaction) {
        this.satisfaction = satisfaction;
    }

    public void evolveSatisfaction() {
        this.satisfaction = Math.pow(Math.pow(satisfaction, 0.5) +
            1, 2);
        /*this.satisfaction += 1;*/
    }
}
```

```java
package general;
import java.util.LinkedList;

import graphics.LocalGraphics;

public class Controller {

    public static int width = 1000; // Largeur en pixels
    public static int height = 1000; // Hauteur en pixels
    public static LocalGraphics graphics;
    public static int numberOfSteps = 100;

    public static Map map;
    public static LinkedList<Drone> drones;
    public static boolean[][] availableCheckpoints;
    public static int[] droneTargets;
    public static double droneSpeed = 10;
    public static double p = 0.9; // Probabilit de faire le choix
       optimal

    public static void init() {
        map = new Map(width, height);
        drones = new LinkedList<Drone>();

        // Ajout de trois checkpoints
        for (int i = 0 ; i < 50 ; i++ ) {map.addCheckpoint(new
           Checkpoint(Math.random() * width * 0.3, Math.random() *
           height * 0.3));}
        for (int i = 0 ; i < 50 ; i++ ) {map.addCheckpoint(new
           Checkpoint(Math.random() * width * 0.3 + width * 0.7,
           Math.random() * height * 0.3));}
        for (int i = 0 ; i < 50 ; i++ ) {map.addCheckpoint(new
           Checkpoint(Math.random() * width * 0.3, Math.random() *
           height * 0.3 + height * 0.7));}
        for (int i = 0 ; i < 50 ; i++ ) {map.addCheckpoint(new
           Checkpoint(Math.random() * width * 0.3 + width * 0.7,
           Math.random() * height * 0.3 + height * 0.7));}
        /*map.addCheckpoint(new Checkpoint(100, 100));
        map.addCheckpoint(new Checkpoint(200, 800));
        map.addCheckpoint(new Checkpoint(200, 700));
        map.addCheckpoint(new Checkpoint(300, 720));
        map.addCheckpoint(new Checkpoint(800, 400));
        map.addCheckpoint(new Checkpoint(700, 500));
        map.addCheckpoint(new Checkpoint(730, 580));
        map.addCheckpoint(new Checkpoint(400, 20));
        map.addCheckpoint(new Checkpoint(500, 200));
        map.addCheckpoint(new Checkpoint(430, 110));
        map.addCheckpoint(new Checkpoint(900, 900));
        map.addCheckpoint(new Checkpoint(950, 830));*/
```

```
        // Ajout de trois drones
        for (int i = 0 ; i < 4 ; i++) {drones.add(new
            Drone(Math.random() * width, Math.random() * height));}

        //
        availableCheckpoints = new
            boolean[drones.size()][map.getCheckpoints().size()];
        for (int i = 0 ; i < drones.size() ; i++) {
            for (int j = 0 ; j < map.getCheckpoints().size() ; j++) {
                availableCheckpoints[i][j] = true;
            }
        }

        droneTargets = new int[drones.size()];
        for (int i = 0 ; i < drones.size() ; i++) droneTargets[i] =
            -1; // Aucun drone n'a de cible

        for (int i = 0 ; i < drones.size() ; i++) makeDecision(i);
            // Intialisation de la d cision pour le drone

        graphics = new LocalGraphics(width, height);
        graphics.draw(drones, map);
    }

    public static void makeDecision(int droneIndex) {
        // Fait prendre au drone une d cision
        // Le drone une probabilit p de choisir le checkpoint le
            plus i n t ressant, et 1-p d'en choisir un au hasard
        int indexMaxInterest = -1;
        double maxInterest = 0;
        double interest;
        for (int j = 0 ; j < map.getCheckpoints().size() ; j++) { //
            On r cup le checkpoint le plus i n t ressant
            if (availableCheckpoints[droneIndex][j]) {
                interest =
                    map.getCheckpoints().get(j).getSatisfaction() /
                    (drones.get(droneIndex).getPosition().distance(map.getCheckpo
                    + 1);
                if (interest > maxInterest) {
                    indexMaxInterest = j;
                    maxInterest = interest;
                }
            }
        }
        Checkpoint checkpointMaxInterest;
        double random = Math.random();
        if (random < p) { // On choisit le checkpoint le plus
            i n t ressant
```

```
            checkpointMaxInterest =
                map.getCheckpoints().get(indexMaxInterest);
        }
        else { // On choisit un checkpoint au hasard
            int index = (int)(Math.random() *
                map.getCheckpoints().size());
            checkpointMaxInterest = map.getCheckpoints().get(index);
        }
        for (int i = 0 ; i < drones.size() ; i++) {
            if (droneTargets[i] == indexMaxInterest & i !=
                droneIndex) { // On est dans le cas d'un conflit
                 if
                    (drones.get(droneIndex).getPosition().distance(checkpointMaxI
                    >
                    drones.get(i).getPosition().distance(checkpointMaxInterest.ge
                     // Le conflit est perdu, on imine ce
                        checkpoint des checkpoints autoriss et on
                        retente
                     availableCheckpoints[droneIndex][indexMaxInterest]
                        = false;
                     makeDecision(droneIndex);
                     return;
                 }
                 else { // Le conflit est gagn, on fait plutt
                     choisir le drone d'indice i
                     droneTargets[droneIndex] = indexMaxInterest;
                     drones.get(droneIndex).setTarget(checkpointMaxInterest);
                     availableCheckpoints[i][indexMaxInterest] =
                        false;
                     makeDecision(i);
                     return;
                 }
            }
        }
        drones.get(droneIndex).setTarget(checkpointMaxInterest);
        droneTargets[droneIndex] = indexMaxInterest;
        return;
    }

    public static void evolve() {
        // La fonction evolve augmente toutes les satisfactions des
           checkpoints, fait bouger les drones vers leur cible, et
           met  jour cette cible si ils l'atteignent
        // La  s p cificit de cette fonction est de fairte prendre
           une  d cision  au drone uniquement quand il est en conflit
           ou quand il atteint un checkpoint
        for (Checkpoint checkpoint : map.getCheckpoints()) {
            checkpoint.evolveSatisfaction();
        }
```

```
        for (int i = 0 ; i < drones.size() ; i++) {
            int checkpointIndex = droneTargets[i];
            boolean isOnTarget = drones.get(i).moveToTarget();
            if (isOnTarget) {
                for (int j = 0 ; j < drones.size() ; j++)
                    availableCheckpoints[j][checkpointIndex] = true;
                map.getCheckpoints().get(droneTargets[i]).setSatisfaction(1);
                makeDecision(i);
            }
        }
        for (int i = 0 ; i < drones.size() ; i++) {makeDecision(i);}
    }

    public static double getTotalSatisfaction() {
        double result = 0;
        for (Checkpoint checkpoint: map.getCheckpoints()) {
            result += checkpoint.getSatisfaction();
        }
        return result / map.getCheckpoints().size();
    }

    public static void printTargets() {
        for (int i = 0 ; i < drones.size() ; i++) {
            System.out.print(droneTargets[i] + "-");
        }
        System.out.println();
    }

    public static void main(String[] args) {
        init();
        while (graphics.frame.isActive()) {
            graphics.draw(drones, map);
            try {Thread.sleep(20);} catch (InterruptedException e) {}
            evolve();
        }
    }
}
```

```
package general;

public class Drone {

    private Vector position;
    private Checkpoint target;

    public Drone(double x, double y) {
        this.position = new Vector(x, y);
    }
```

```
    public Vector getPosition() {
        return position;
    }

    public void setPosition(Vector position) {
        this.position = position;
    }

    public Checkpoint getTarget() {
        return target;
    }

    public void setTarget(Checkpoint target) {
        this.target = target;
    }

    public boolean moveToTarget() {
        if (target.getPosition().distance(position) <
           Controller.droneSpeed) {
            this.position = target.getPosition().copy();
            return true;
        }
        Vector speedVector =
           target.getPosition().remove(position).normalize(Controller.droneSpeed
        position = position.add(speedVector);
        return false;
    }
}
```

```
package general;
import java.util.LinkedList;

public class Map {

    private double width;
    private double height;

    private LinkedList<Checkpoint> checkpoints;

    public Map(double width, double height) {
        this.width = width;
        this.height = height;
        this.checkpoints = new LinkedList<>();
    }

    public void addCheckpoint(Checkpoint checkpoint) {
        checkpoints.add(checkpoint);
    }
```

```
    public double getWidth() {
        return width;
    }

    public void setWidth(double width) {
        this.width = width;
    }

    public double getHeight() {
        return height;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    public LinkedList<Checkpoint> getCheckpoints() {
        return checkpoints;
    }

    public void setCheckpoints(LinkedList<Checkpoint> checkpoints) {
        this.checkpoints = checkpoints;
    }


}
```

```
package general;

public class Target {

}
```

```
package general;

public class Vector {

    private double x;
    private double y;

    public Vector(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() {
        return x;
    }
```

```
    public void setX(double x) {
        this.x = x;
    }

    public double getY() {
        return y;
    }

    public void setY(double y) {
        this.y = y;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Vector other = (Vector) obj;
        if (Double.doubleToLongBits(x) !=
           Double.doubleToLongBits(other.x))
            return false;
        if (Double.doubleToLongBits(y) !=
           Double.doubleToLongBits(other.y))
            return false;
        return true;
    }

    public double distance(Vector vector) {
        return Math.pow(Math.pow(this.x - vector.x, 2) +
           Math.pow(this.y - vector.y, 2), 0.5);
    }

    public Vector add(Vector vector) {
        return new Vector(vector.x + this.x, vector.y + this.y);
    }

    public Vector remove(Vector vector) {
        return new Vector(- vector.x + this.x, - vector.y + this.y);
    }

    public Vector multiply(double coeff) {
        return new Vector(this.x * coeff, this.y * coeff);
    }

    public boolean isInBounds() {
        return (x >= 0 & y >= 0 & x <= Controller.width & y <=
```

```
            Controller.height);
    }

    public double norm() {
        return Math.pow(Math.pow(this.x, 2) + Math.pow(this.y, 2),
            0.5);
    }

    public Vector normalize(double norm) {
        return this.multiply(norm / this.norm());
    }

    public Vector copy() {
        return new Vector(this.x, this.y);
    }
}
```

2.2 Package graphics

```
package graphics;

import javax.swing.JFrame;

// Cadre de l'objet graphique
public class Frame extends JFrame{

    private static final long serialVersionUID = 1L;

    // Champ fenetre
    public Window window;

    // Constructeur d'un cadre a partir de sa largeur, sa hauteur et
       de la grille grid. Dele au constructeur de Window
    public Frame(int frameWidth, int frameHeight) {
        this.setTitle("Simulation");
        this.window = new Window(frameWidth, frameHeight);
        this.setSize(frameWidth, frameHeight);
        this.getContentPane().add(window);
        this.addKeyListener(window);
        this.addMouseListener(window);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setResizable(false);
        this.setLocationRelativeTo(null);
        this.setLayout(null);
        this.setVisible(true);
    }
}
```

```java
package graphics;

import java.awt.Color;
import java.awt.Graphics;
import java.awt.event.KeyEvent;
import java.awt.event.MouseEvent;
import java.util.LinkedList;

import general.Checkpoint;
import general.Drone;
import general.Map;

// Objet  construire pour obtenir un rendu graphique
public class LocalGraphics {

    // Largeur et hauteur en pixels de la  f netre
    public int frameWidth;
    public int frameHeight;

    // Champs graphiques
    public Frame frame;
    public Window window;
    public Graphics graphics;

    public int checkpointSize = 11;
    public int droneSize = 10;

    // Construit un affichage de n*m cases, tileWidth*tileHeight
       pixels,  partir d'une grille grid
    public LocalGraphics(int width, int height) {
        this.frameWidth = (int)(width + 50);
        this.frameHeight = (int)(height + 50);

        this.frame = new Frame(frameWidth, frameHeight);
        this.window = frame.window;
        this.graphics = window.getGraphics();
        try {
            Thread.sleep(100);
        } catch (InterruptedException e) {
            System.out.println("Interruption impromptue");
        }
        window.localGraphics = this;
    }

    public void draw(LinkedList<Drone> drones, Map map) {
        graphics.setColor(Color.WHITE);
        graphics.fillRect(0, 0, frameWidth, frameHeight);
        // Checkpoints
        graphics.setColor(Color.BLACK);
```

```
        for (Checkpoint checkpoint : map.getCheckpoints())
            graphics.drawOval((int)checkpoint.getPosition().getX(),
            (int)checkpoint.getPosition().getY(), checkpointSize,
            checkpointSize);
        // Drones
        graphics.setColor(Color.GREEN);
        for (Drone drone : drones)
            graphics.fillOval((int)drone.getPosition().getX(),
            (int)drone.getPosition().getY(), droneSize, droneSize);


    }

    // Change la grille en fonction des deux paramtres writingType
       et writingContainer
    public void onMouseClicked(MouseEvent ev) {

    }

    // Rgle le type de dessin de mouseState : e->EMPTY/EMPTY,
       d->EMPTY/DRONE, o->OBSTACLE/EMPTY, x->EMPTY/ENNEMY,
       c->CHECKPOINT/EMPTY
    public void keyTyped(KeyEvent ev) {
    }

    // Met  jour l'affichage
    public void refreshGraphics() {

    }
}
```

```
package graphics;

import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.MouseEvent;
import java.awt.event.MouseListener;

import javax.swing.JPanel;

// Objet fentre pour l'affichage graphique, ragit aux actions du
   lavier et de la souris
public class Window extends JPanel implements KeyListener,
   MouseListener{

    private static final long serialVersionUID = 1L;

    // Objet graphique appelant
    public LocalGraphics localGraphics;
```

```
// Constructeur   partir d'une grille
public Window(int windowWidth, int windowHeight) {
    super();
    this.setOpaque(false);
    this.setBounds(0, 0, windowWidth, windowHeight);
}

// Apelle la fonction de localGraphics qui  r gle  le type de
   dessin
@Override
public void keyTyped(KeyEvent ev) {
    localGraphics.keyTyped(ev);
}

// M thode  de l'interface KeyListener
@Override
public void keyPressed(KeyEvent ev) {
}

// M thode  de l'interface KeyListener
@Override
public void keyReleased(KeyEvent ev) {
}

// M thode  de l'interface MouseListener
@Override
public void mouseClicked(MouseEvent me) {
}

// Appelle la fonction onMouseClicked de l'objet graphique pour
   effectuer l'action sur la tule clique en fonction de l' at
      s   lionn de la souris
@Override
public void mousePressed(MouseEvent me) {
    localGraphics.onMouseClicked(me);
}

// M thode  de l'interface MouseListener
@Override
public void mouseReleased(MouseEvent me) {
}

// M thode  de l'interface MouseListener
@Override
public void mouseEntered(MouseEvent me) {
}

// M thode  de l'interface MouseListener
```

```
    @Override
    public void mouseExited(MouseEvent me) {
    }

}
```

3 Méthode par potentiel

La méthode utlise la bibliothèque VCL, fournie dans le cadre du cours INF443, qui ne figure pas ici.

```
#include "structure.hpp"

using namespace vcl;

// Graphical constants
float arm_length = 1.0f;
float arm_radius = 0.1f;
float rotor_radius_major = 0.3f;
float rotor_radius_minor = 0.05f;
float scale = 0.02f;
float default_drone_height = 0.1f;

float Drone::size = 0.02f;
float Drone::speed = 0.003f;
int Drone::direction_samples = 50;
int Drone::measures_per_sample = 10;
float Drone::sample_distance = 0.3f;
int Drone::frames_per_step = 10;

int counter = 0;

Drone::Drone()
{
    position = vec3(rand_interval(), rand_interval(),
       default_drone_height);
    direction = 2 * pi * rand_interval();
}

void Drone::update_position(Terrain *terrain)
{
    assert(positions.x >= 0 & position.x <= 1 & position.y >= 0 &
       position.y <= 1);

    if (counter == frames_per_step) {
        float score;
        float best_score = 0.1 * measures_per_sample;
        float best_direction = 0;
        float local_direction = 0;
        for (int i = 0; i <= direction_samples; i++) {
            local_direction = -pi + 2 * float(i) / direction_samples
               * pi;
            score = evaluate_direction(local_direction, terrain);
            if (score < best_score) {
```

```
                best_score = score;
                best_direction = local_direction;
            }
        }
        direction = best_direction;
        counter = 0;
    }
    else
        counter++;

    position += speed * vec3(cos(direction), sin(direction), 0);
    if (position.x < 0) {
        position.x = -position.x;
    }
    else if (position.x > 1) {
        position.x = 2 - position.x;
    }
    if (position.y < 0) {
        position.y = -position.y;
    }
    else if (position.y > 1) {
        position.y = 2 - position.y;
    }
    for (obstacle _obstacle : terrain->obstacles_list) {
        if (_obstacle.is_in_bounds(position.xy())) {
            position -= speed * vec3(cos(direction), sin(direction),
               0);
            direction = pi - direction;
        }
    }

}

void Drone::update_visual(hierarchy_mesh_drawable* drone_visual)
{


    vec3 const previous_position =
       drone_visual->operator[]("root").transform.translate;
    rotation bird_rotation = rotation_between_vector({ 1, 0, 0 },
       normalize(position - previous_position));

    // Mouvement de l'oiseau
    drone_visual->operator[]("root").transform.translate = position;
    drone_visual->operator[]("root").transform.rotate =
       rotation(bird_rotation);

    drone_visual->update_local_to_global_coordinates();
}
```

```
vcl::vec3 Drone::get_position()
{
    return position;
}

hierarchy_mesh_drawable Drone::get_mesh_drawable()
{

    hierarchy_mesh_drawable result;
    mesh_drawable arm =
       mesh_drawable(mesh_primitive_cubic_grid(vec3(-arm_length / 2,
       -arm_radius / 2, -arm_radius / 2),
        vec3(-arm_length / 2, -arm_radius / 2, arm_radius / 2),
        vec3(-arm_length / 2, arm_radius / 2, arm_radius / 2),
        vec3(-arm_length / 2, arm_radius / 2, -arm_radius / 2),
        vec3(arm_length / 2, -arm_radius / 2, -arm_radius / 2),
        vec3(arm_length / 2, -arm_radius / 2, arm_radius / 2),
        vec3(arm_length / 2, arm_radius / 2, arm_radius / 2),
        vec3(arm_length / 2, arm_radius / 2, -arm_radius / 2)));
    mesh_drawable rotor =
       mesh_drawable(mesh_primitive_torus(rotor_radius_major,
       rotor_radius_minor));
    arm.shading.color = vec3(0.0f, 0.0f, 0.0f);
    result.add(mesh_drawable(), "root");
    result.add(arm, "first_arm", "root");
    result.add(arm, "second_arm", "root");
    result.add(rotor, "first_left_rotor", "first_arm",
       vec3(arm_length / 2 + rotor_radius_major, 0, 0));
    result.add(rotor, "first_right_rotor", "first_arm",
       vec3(-arm_length / 2 - rotor_radius_major, 0, 0));
    result.add(rotor, "second_left_rotor", "second_arm",
       vec3(-arm_length / 2 - rotor_radius_major, 0, 0));
    result.add(rotor, "second_right_rotor", "second_arm",
       vec3(arm_length / 2 + rotor_radius_major, 0, 0));

    result["first_arm"].transform.rotate = rotation(vec3(0, 0, 1),
       pi / 4);
    result["second_arm"].transform.rotate = rotation(vec3(0, 0, 1),
       -pi / 4);

    result["root"].transform.translate = position;
    result["root"].transform.scale = scale;

    result.update_local_to_global_coordinates();

    return result;
}
```

```
double Drone::evaluate_direction(float local_direction, Terrain*
    terrain)
{
    assert(local_direction >= 0 & local_direction <= 2 * pi);

    double result = 0;
    vec2 direction = vec2(cos(local_direction),
        sin(local_direction)) * sample_distance / measures_per_sample;
    vec2 position_copy = vec2(position.x, position.y);

    for (int i = 0; i < measures_per_sample; i++) {
        if (!get_position_direction_after(position_copy, direction,
            false, terrain)) return 0;
        result += terrain->evaluate_terrain_live(position_copy.x,
            position_copy.y);
    }

    return result;
}

bool Drone::get_position_direction_after(vec2& position, vec2&
    direction, bool has_bounced, Terrain* terrain)
{
    position += direction;

    // Checks if it is in bounds
    if (position.x < 0) {
        if (has_bounced) return false;
        has_bounced = true;
        position.x = -position.x;
        direction.x = -direction.x;
    }
    else if (position.x > 1) {
        if (has_bounced) return false;
        has_bounced = true;
        position.x = 2 - position.x;
        direction.x = -direction.x;
    }
    if (position.y < 0) {
        if (has_bounced) return false;
        has_bounced = true;
        position.y = -position.y;
        direction.y = -direction.y;
    }
    else if (position.y > 1) {
        if (has_bounced) return false;
        has_bounced = true;
        position.y = 2 - position.y;
        direction.y = -direction.y;
```

```
    }

    // Checks if it is in an obstacle
    for (obstacle _obstacle : terrain->obstacles_list) {
        if (_obstacle.is_in_bounds(position)) {
            if (has_bounced) return false;
            has_bounced = true;
            direction.x = -direction.x;
            direction.y = -direction.y;
            position += direction;
        }
    }

    return true;
}
```

```
#include "vcl/vcl.hpp"
#include <iostream>
#include <chrono>

#include "structure.hpp"

using namespace vcl;

struct gui_parameters {
    bool display_frame = false;
    bool display_wireframe = false;
};

struct user_interaction_parameters {
    vec2 mouse_prev;
    timer_fps fps_record;
    mesh_drawable global_frame;
    gui_parameters gui;
    bool cursor_on_gui;
};
user_interaction_parameters user;

struct scene_environment
{
    camera_around_center camera;
    mat4 projection;
    vec3 light;
};
scene_environment scene;


void mouse_move_callback(GLFWwindow* window, double xpos, double
    ypos);
```

```
void window_size_callback(GLFWwindow* window, int width, int height);

void initialize_data();
void display_interface();
void update_graphics();

std::vector<Drone> drones;
int number_of_drones = 0;
hierarchy_mesh_drawable drone_visual;

Terrain terrain;
mesh_drawable terrain_current_visual;


int main(int, char* argv[])
{
    std::cout << "Run " << argv[0] << std::endl;

    int const width = 1280, height = 1024;
    GLFWwindow* window = create_window(width, height);
    window_size_callback(window, width, height);
    std::cout << opengl_info_display() << std::endl;;

    imgui_init(window);
    glfwSetCursorPosCallback(window, mouse_move_callback);
    glfwSetWindowSizeCallback(window, window_size_callback);

    std::cout<<"Initialize data ..."<<std::endl;
    initialize_data();

    std::cout<<"Start animation loop ..."<<std::endl;
    user.fps_record.start();
    glEnable(GL_DEPTH_TEST);

    while (!glfwWindowShouldClose(window))
    {
        scene.light = scene.camera.position();
        user.fps_record.update();

        glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
        glClear(GL_COLOR_BUFFER_BIT);
        glClear(GL_DEPTH_BUFFER_BIT);
        imgui_create_frame();
        if(user.fps_record.event) {
            std::string const title = "VCL Display - "
                +str(user.fps_record.fps)+" fps";
            glfwSetWindowTitle(window, title.c_str());
        }
```

```
        ImGui::Begin("GUI",NULL,ImGuiWindowFlags_AlwaysAutoResize);
        user.cursor_on_gui = ImGui::IsAnyWindowFocused();

        if(user.gui.display_frame) draw(user.global_frame, scene);

        display_interface();
        update_graphics();

        ImGui::End();
        imgui_render_frame(window);
        glfwSwapBuffers(window);
        glfwPollEvents();
    }
    imgui_cleanup();
    glfwDestroyWindow(window);
    glfwTerminate();

    return 0;
}

void update_graphics()
{
    // Update des objets
    for (int i = 0; i < number_of_drones; i++) {
        drones[i].update_position(&terrain);
        drones[i].update_visual(&drone_visual);
        draw(drone_visual, scene);
    }

    terrain.update_potential(&drones, terrain_current_visual);
    draw(terrain_current_visual, scene);

    if (user.gui.display_wireframe) {
        draw_wireframe(terrain_current_visual, scene, { 0, 0, 1 });
    }
}

void initialize_data()
{

    // Basic setups of shaders and camera
    GLuint const shader_mesh =
       opengl_create_shader_program(opengl_shader_preset("mesh_vertex"),
       opengl_shader_preset("mesh_fragment"));
    mesh_drawable::default_shader = shader_mesh;
    mesh_drawable::default_texture =
       opengl_texture_to_gpu(image_raw{1,1,image_color_type::rgba,{255,255,255,2

    user.global_frame = mesh_drawable(mesh_primitive_frame());
```

```
    user.gui.display_frame = false;
    scene.camera.distance_to_center = 2.5f;
    scene.camera.look_at({-3,1,2}, {0,0,0.5}, {0,0,1});

    for (int i = 0; i < number_of_drones; i++)
        drones.push_back(Drone());

    drone_visual = Drone().get_mesh_drawable();

    terrain_current_visual =
        mesh_drawable(terrain.get_current_mesh());

}

void display_interface()
{
    ImGui::Checkbox("Frame", &user.gui.display_frame);
    ImGui::Checkbox("Wireframe", &user.gui.display_wireframe);

}

void window_size_callback(GLFWwindow* , int width, int height)
{
    glViewport(0, 0, width, height);
    float const aspect = width / static_cast<float>(height);
    scene.projection = projection_perspective(50.0f*pi/180.0f,
        aspect, 0.1f, 100.0f);
}

void mouse_move_callback(GLFWwindow* window, double xpos, double
    ypos)
{
    vec2 const  p1 = glfw_get_mouse_cursor(window, xpos, ypos);
    vec2 const& p0 = user.mouse_prev;
    glfw_state state = glfw_current_state(window);

    auto& camera = scene.camera;
    if(!user.cursor_on_gui){
        if(state.mouse_click_left && !state.key_ctrl)
            scene.camera.manipulator_rotate_trackball(p0, p1);
        if(state.mouse_click_left && state.key_ctrl)
            camera.manipulator_translate_in_plane(p1-p0);
        if(state.mouse_click_right)
            camera.manipulator_scale_distance_to_center( (p1-p0).y );
    }

    user.mouse_prev = p1;
}
```

```
void opengl_uniform(GLuint shader, scene_environment const&
    current_scene)
{
    opengl_uniform(shader, "projection", current_scene.projection);
    opengl_uniform(shader, "view", scene.camera.matrix_view());
    opengl_uniform(shader, "light", scene.light, false);
}
```

```
#pragma once

#include "vcl/vcl.hpp"

struct hill {
    vcl::vec2 center;
    float sigma;
    float height;
};

struct obstacle {
    vcl::vec2 center;
    float radius;
    bool is_in_bounds(vcl::vec2 position);
};

class Terrain;

class Drone {

public:
    static float size;
    static float speed;
    static int frames_per_step;
    static int direction_samples;
    static int measures_per_sample;
    static float sample_distance;
    static float surveillance_radius;

    Drone();
    vcl::hierarchy_mesh_drawable get_mesh_drawable();
    void update_position(Terrain* terrain);
    void update_visual(vcl::hierarchy_mesh_drawable* drone_visual);
    vcl::vec3 get_position();

private:
    float direction;
    vcl::vec3 position;
    double Drone::evaluate_direction(float direction, Terrain*
        terrain);
```

```
    bool get_position_direction_after(vcl::vec2& position,
        vcl::vec2& direction, bool has_bounced, Terrain* terrain);
};

class Drone;

class Terrain {

private:
    int n, m;
    std::vector<hill> hills_list;
    vcl::mesh current_potential;
    vcl::buffer<int> obstacles;
    vcl::mesh initial_potential;
    void initialize_terrain();
    vcl::vec3 evaluate_terrain(float x, float y);
    vcl::vec3 get_color_from_height(int i, int j);

public:

    Terrain();
    std::vector<obstacle> obstacles_list;
    void update_potential(std::vector<Drone> *drones,
        vcl::mesh_drawable& terrain_visual);
    vcl::mesh get_initial_mesh();
    vcl::mesh get_current_mesh();
    float Terrain::evaluate_terrain_live(float x, float y);
};
```

```
#include "structure.hpp"

using namespace vcl;

float potential_min = -0.2f;
float potential_max = 0.03f;
float potential_decrease = 0.0002f;
float potential_value_at_reset = 0.00f;
float obstacle_potential = 0.1f;
float drone_surveillance_radius = 0.05f;

// Colors
vec3 red = vec3(1.0f, 0.0f, 0.0f);
vec3 green = vec3(0.0f, 1.0f, 0.0f);

Terrain::Terrain() {
    n = 50;
    m = 50;

    // Cration des colines
```

```
    hill hill_0{ vec2(0.5, 0.8), 0.1, 0.2 };
    hill hill_1{ vec2(0.2, 0.5), 0.4, 0.5 };
    hill hill_2{ vec2(0.1, 0.1), 0.2, 0.2 };
    hills_list.push_back(hill_1);
    hills_list.push_back(hill_0);
    hills_list.push_back(hill_2);

    // Cration des obstacles
    obstacle obstacle_1{ vec2(0.2, 0.9), 0.1 };
    obstacle obstacle_2{ vec2(0.4, 0.4), 0.1 };
    obstacle obstacle_3{ vec2(0.2, 0.2), 0.1 };
    obstacle obstacle_4{ vec2(0.9, 0.2), 0.15 };
    obstacles_list.push_back(obstacle_1);
    obstacles_list.push_back(obstacle_2);
    obstacles_list.push_back(obstacle_3);
    obstacles_list.push_back(obstacle_4);

    obstacles.resize((n + 1) * (m + 1));
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            for (obstacle _obstacle : obstacles_list) {
                if (_obstacle.is_in_bounds(vec2(float(i) / n,
                   float(j) / m)))
                    obstacles[i * (m + 1) + j] = true;
            }
        }
    }

    // Configuration, en bleu, du potentiel initial
    initialize_terrain();
}

void Terrain::update_potential(std::vector<Drone> *drones,
   mesh_drawable& current_potential_visual)
{
    float x, y, z;
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            if (obstacles[i * (m + 1) + j]) continue;
            x = float(i) / n;
            y = float(j) / m;
            z = ((current_potential.position[i * (m + 1) + j] +
               potential_decrease * initial_potential.position[i *
               (m + 1) + j]) / (1 + potential_decrease)).z;
            current_potential.position[i * (m + 1) + j].z = z;
            current_potential.color[i * (m + 1) + j] =
               get_color_from_height(i, j);

            for (int k = 0; k < drones->size(); k++) {
```

```
                if (norm(vec2(x, y) -
                    drones->operator[](k).get_position().xy()) <
                    drone_surveillance_radius) {
                    current_potential.position[i * (m + 1) + j].z =
                        potential_value_at_reset;
                    current_potential.color[i * (m + 1) + j] =
                        vec3(0.0f, 1.0f, 0.0f);
                }
            }
        }
    }

    current_potential.compute_normal();

    current_potential_visual.update_position(current_potential.position);
    current_potential_visual.update_normal(current_potential.normal);
    current_potential_visual.update_color(current_potential.color);
}

mesh Terrain::get_initial_mesh()
{
    return initial_potential;
}

mesh Terrain::get_current_mesh()
{
    return current_potential;
}

float Terrain::evaluate_terrain_live(float x, float y)
{

    int i = int(x * n);
    int j = int(y * m);

    float result = 0;

    result += current_potential.position[i * (m + 1) + j].z;
    result += current_potential.position[(i + 1) * (m + 1) + j].z;
    result += current_potential.position[i * (m + 1) + j + 1].z;
    result += current_potential.position[(i + 1) * (m + 1) + j +
        1].z;

    return result / 4;
}

vec3 Terrain::evaluate_terrain(float x, float y)
{
    if (x < 0 || x > 1 || y < 0 || y > 1) return vec3(x, y, 0);
```

```
    float z = 0;
    hill _hill;
    for (int i = 0; i < hills_list.size(); i++) {
        _hill = hills_list[i];
        z -= _hill.height * exp(-pow(norm(vec2(x, y) - _hill.center)
           / _hill.sigma, 2));
    }

    return vec3(x, y, z);
}


vec3 Terrain::get_color_from_height(int i, int j)
{
    if (obstacles[i * (m + 1) + j] == true) return vec3(0, 0, 0);
    float t = (current_potential.position[i * (m + 1) + j].z -
       potential_min) / (potential_max - potential_min);
    return t * green + (1 - t) * red;
}

void Terrain::initialize_terrain()
{
    initial_potential.position.resize((n + 1) * (m + 1));
    initial_potential.color.resize((n + 1) * (m + 1));
    initial_potential.connectivity.resize(2 * n * m);


    current_potential.position.resize((n + 1) * (m + 1));
    current_potential.color.resize((n + 1) * (m + 1));
    current_potential.connectivity.resize(2 * n * m);

    float x, y;
    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            x = float(i) / n;
            y = float(j) / m;
            initial_potential.position[i * (m + 1) + j] =
               evaluate_terrain(x, y);
            current_potential.position[i * (m + 1) + j] =
               evaluate_terrain(x, y);
            if (obstacles[i * (m + 1) + j])
               current_potential.position[i * (m + 1) + j].z =
               obstacle_potential;
        }
    }
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < m; j++) {
            initial_potential.connectivity[2 * (i * m + j)] =
```

```
                uint3(i * (m + 1) + j, (i + 1) * (m + 1) + j, i * (m
                + 1) + j + 1);
            initial_potential.connectivity[2 * (i * m + j) + 1] =
                uint3((i + 1) * (m + 1) + j, (i + 1) * (m + 1) + j +
                1, i * (m + 1) + j + 1);
            current_potential.connectivity[2 * (i * m + j)] =
                uint3(i * (m + 1) + j, (i + 1) * (m + 1) + j, i * (m
                + 1) + j + 1);
            current_potential.connectivity[2 * (i * m + j) + 1] =
                uint3((i + 1) * (m + 1) + j, (i + 1) * (m + 1) + j +
                1, i * (m + 1) + j + 1);
        }
    }

    for (int i = 0; i <= n; i++) {
        for (int j = 0; j <= m; j++) {
            initial_potential.color[i * (m + 1) + j] =
                get_color_from_height(i, j);
            current_potential.color[i * (m + 1) + j] =
                get_color_from_height(i, j);
        }
    }

    initial_potential.fill_empty_field();
    current_potential.fill_empty_field();
}

bool obstacle::is_in_bounds(vcl::vec2 position)
{
    return (norm(position - center) < radius);
}
```

4 Méthodes de tests

Les classes Java et c++ qui collectent les données et effectuent les tests.

```
#include "vcl/vcl.hpp"
#include <chrono>
#include "structure.hpp"
#include <iostream>
#include <cstdio>

using namespace std;

const int time_limit = 10000;

// constructeur
```

```
Statistics::Statistics() {
    int actual_size = 0;
    double list_scores [time_limit];
    double list_dist[time_limit];
    double list_avg_dist[time_limit];
}

// afficheur des scores
int Statistics::update_scores(double nbre) {
    if (actual_size < time_limit) { // tant que le tableau n'est pas
       plein on le remplit
        list_scores[actual_size] = nbre;
        ++actual_size;
    }

    if (actual_size == time_limit) { // une fois qu'il est plein on
        g n  un fichier txt
        ofstream myfile;
        myfile.open("scores_mouvement_3d_cpp.txt");
        myfile << time_limit << std::endl;
        // std::cout << "time_limit" << time_limit << std::endl;
        for (unsigned i = 0; i < time_limit; i++) {
            myfile << list_scores[i] << "\n";
        }
        //std::cout << actual_size << std::endl;
        myfile.close();

    }
        return 0.0;
}

// afficheur des distances d'un drone  une position en fonction du
   temps
int Statistics::update_dist(double nbre) {
    if (actual_size < time_limit) { // tant que le tableau n'est pas
       plein on le remplit
        list_dist[actual_size] = nbre;
        ++actual_size;
    }
    std::cout << "size : " << actual_size << std::endl;

    if (actual_size == time_limit) { // une fois qu'il est plein on
        g n  un fichier txt
        ofstream myfile;
        myfile.open("dist_mouvement_3d_cpp.txt");
        myfile << time_limit << std::endl;

        for (unsigned i = 0; i < time_limit; i++) {
            myfile << list_dist[i] << "\n";
```

```
        }
        //std::cout << actual_size << std::endl;
        myfile.close();

    }
    return 0.0;
}

// afficheur des distances d'un drone  une position en fonction du
   temps
int Statistics::update_avg_dist(double nbre) {
    if (actual_size < time_limit) { // tant que le tableau n'est pas
       plein on le remplit
        list_avg_dist[actual_size] = nbre;
        ++actual_size;
    }
    std::cout << "size : " << actual_size << std::endl;

    if (actual_size == time_limit) { // une fois qu'il est plein on
        g n  un fichier txt
        ofstream myfile;
        myfile.open("dist_avg_mouvement_3d_cpp.txt");
        myfile << time_limit << std::endl;

        for (unsigned i = 0; i < time_limit; i++) {
            myfile << list_avg_dist[i] << "\n";
        }
        //std::cout << actual_size << std::endl;
        myfile.close();

    }
    return 0.0;
}
```

```
package general;

import java.io.IOException;
import java.util.ArrayList;
import java.util.LinkedList;

public class Statistics {
    public static ArrayList<Double> avgScoring;
    public static int timeLimit;
    public static ArrayList<ArrayList<Double>> avgDist;
    public static ArrayList<Double> worstCheckpoint;

    /**
     *
     * @param duree : nombre d'iterations durant lequel on veut
```

```
     recuperer les donnees, definit la taille des fichiers txt
     qui seront generes
 */
public Statistics(int duree) {
    avgScoring = new ArrayList<Double>();
    timeLimit = duree;
    avgDist = new ArrayList<ArrayList<Double>>();
    worstCheckpoint = new ArrayList<Double>();
}

/**
 * Ajoute la valeur donnee  la liste des scores de satisfaction
 * @param score : valeur de la satisfaction qu'on souhaite
     ajouter  la liste des scores
 * @throws IOException
 */
public void scoreUpdate(double score) throws IOException{
    avgScoring.add(score);
    if(avgScoring.size()== timeLimit)
       TxtGenerator.scoreUpdate(avgScoring);
}

 /**
 * Ajoute la le n-uplet donnee  la liste de n-uplet des
     distances moyennes. Un n-uplet correspond  un instant t.
 * @param drones
 * @throws IOException
 */
public void avgDistance(LinkedList<Drone> drones) throws
   IOException {
    ArrayList<Double> a = new ArrayList<>(drones.size());
    for(int i = 0; i<drones.size(); i++) {
        a.add(drones.get(i).getAvgDistToOthers(drones));
    }
    avgDist.add(a);
    System.out.println(avgDist.size());
    if(avgDist.size()== timeLimit)
       TxtGenerator.avgDistUpdate(avgDist);

}

/**
 * Ajoute la valeur donnee  la liste des scores des pires
     insatisfactions
 * @param drones
 * @throws IOException
 */
public void worstUpdate(double score) throws IOException{
    worstCheckpoint.add(score);
```

```
        if(worstCheckpoint.size()== timeLimit)
            TxtGenerator.worstUpdate(worstCheckpoint);
    }
}
```

```
package general;
import java.io.*;
import java.util.ArrayList;

class TxtGenerator
{
    // Genere un fichier txt d'une longueur Time_limit
    // contenant la satisfaction globale de la map en fonction du
       temps
    public static void scoreUpdate(ArrayList<Double> satisfaction)
       throws IOException
    {
        BufferedWriter tampon = new BufferedWriter(new
           FileWriter("satisfaction.txt"));
        PrintWriter sortie = new PrintWriter(tampon);
        sortie.println(satisfaction.size());
        for (int i = 0; i < satisfaction.size(); i++)
        {
            sortie.println(satisfaction.get(i));
        }
        sortie.flush();
        sortie.close();
    }

    // Genere un fichier txt d'une longueur Time_limit
    // contenant la la distance moyenne des drones entre eux en
       fonction du temps
    // = permet d'etudier la faon de voler (essaim, solitaire,
       petits groupes)
    public static void avgDistUpdate (ArrayList<ArrayList<Double>>
       distMatrix) throws IOException{
        BufferedWriter tampon = new BufferedWriter(new
           FileWriter("dist.txt"));
        PrintWriter sortie = new PrintWriter(tampon);
        sortie.println(distMatrix.size());
        sortie.println(distMatrix.get(0).size());
        for(ArrayList<Double> a : distMatrix) {
            for(int i = 0; i<a.size(); i++) {
                sortie.println(a.get(i));
            }
        }
        sortie.flush();
        sortie.close();
    }
```

```
    // Genere un fichier txt d'une longueur Time_limit
    // contenant la pire satisfaction sur la map en fonction du temps
    public static void worstUpdate(ArrayList<Double> satisfaction)
       throws IOException
    {
        BufferedWriter tampon = new BufferedWriter(new
           FileWriter("worst.txt"));
        PrintWriter sortie = new PrintWriter(tampon);
        sortie.println(satisfaction.size());
        for (int i = 0; i < satisfaction.size(); i++)
        {
            sortie.println(satisfaction.get(i));
        }
        sortie.flush();
        sortie.close();
    }
}
```