université
PARIS-SACLAY

FLAPPY BIRD
RAPPORT DE TUTORIEL

mis en page par

Xue YANG

(22200431)

Licence 3 Informatique parcours Miage

Groupe 3

Janvier 2023

UE : PCII

1. Introduction

L'objectif de ce projet est de développer un jeu inspiré de "Flappy Bird" en utilisant le langage de programmation Java orienté objet. Ce projet fait partie de l'Unité d'Enseignement Programmation Concurrentielle et Interfaces Interactives en troisième année de Licence Informatique, et vise à familiariser les étudiants avec les éléments clés de la PCII.

L'introduction de ce jeu est la suivante : Dans ce jeu, vous contrôlez un oiseau représenté par une forme ovale. Vous devez naviguer le long d'une ligne générée automatiquement en volant vers le haut et vers le bas en évitant de quitter la ligne. Les joueurs peuvent faire monter l'oiseau en cliquant sur l'écran, mais il redescendra de lui-même. Le score du joueur est basé sur la longueur du parcours effectué dans le jeu et sera affiché à la fin du jeu grâce à la classe JOptionPane. Si l'oiseau quitte la ligne, les threads de vol et d'avancement des obstacles s'arrêteront également.

Voici à quoi pourrait ressembler l'interface graphique de ce jeu:

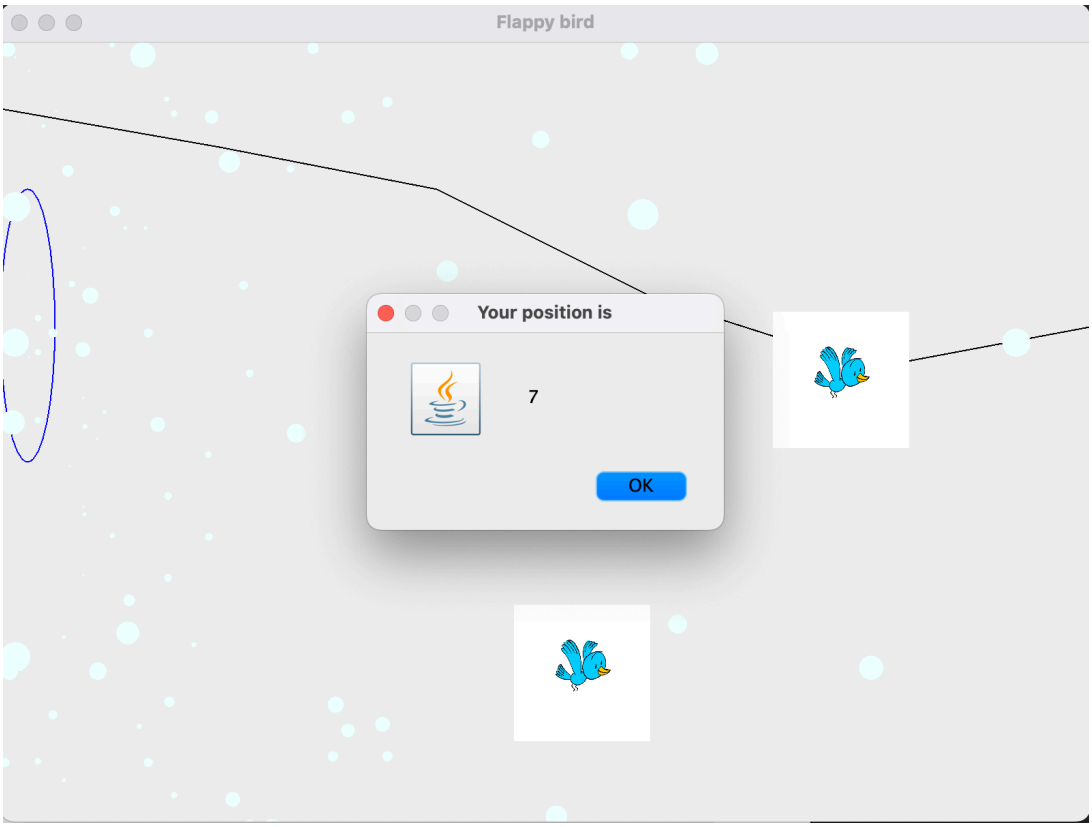

Figure 1 – Capture d'écran de l'interface à la fin du jeu

Dans ce rapport, je vais vous décrire les étapes prises pour accomplir le projet, certains aspects du processus de développement, ainsi que le fonctionnement du jeu.

2. Analyse globale

Voici les fonctionnalités principales à mettre en œuvre dans le jeu :

1、L'interface graphique : Créer une fenêtre d'interface graphique pour afficher le jeu.

2、Ovale : Dessiner un ovale sur l'interface pour représenter l'oiseau contrôlé par le joueur.

3、Saut et événements souris : Écrire une méthode de saut qui permet au joueur de faire sauter l'oiseau en cliquant avec la souris sur l'interface.

4、Vol : Faire en sorte que l'oiseau puisse tomber à une vitesse déterminée par défaut.

5、Génération de points de trajet de vol: En utilisant une méthode de génération aléatoire, des points nécessaires pour composer le trajet de vol sont générés, et leur pente est assurée d'être appropriée pour que les joueurs puissent voler le long du trajet.

6、Avancement de la ligne : Générer une ligne qui se déplace automatiquement vers la gauche.

7、Détection de collision : Dès que l'ovale du joueur sort de la ligne, arrêter immédiatement le mouvement de l'ovale et de la ligne.

8、Boîte de message : Dès que le jeu s'arrête, afficher une boîte de message avec le score du joueur.

9、Image d'oiseau : Charger l'image d'oiseau fournie dans le jeu et la faire voler de manière dynamique.

10、Bruit de Perlin : Utiliser l'algorithme de bruit de Perlin pour simuler des nuages dynamique.

Après une analyse approfondie des fonctionnalités du jeu, je pense que la mise en œuvre de la détection de collision et de la génération de points de trajet de vol sera difficile. La détection de collision nécessite une précision mathématique élevée ainsi qu'une bonne compréhension de la gestion des collisions entre des formes géométriques complexes dans un environnement de jeu en temps réel. De même, la génération de points de trajet de vol nécessite une bonne compréhension des algorithmes de génération aléatoire et de garantir que les points générés soient appropriés pour les trajets de vol. Une mise en œuvre incorrecte de ces deux fonctionnalités peut affecter l'expérience de jeu pour les joueurs et la qualité graphique du jeu. Il est donc crucial de consacrer suffisamment de temps et de ressources pour s'assurer que la détection de collision et la génération de points de trajet de vol soient implémentées de manière efficace.

3. Plan de développement

Séance 1

- Liste des tâches :
- Analyse du problème (15 mn)
- Conception, développement et test d'une fenêtre avec un ovale (30 mn)
- Conception, développement et test du mécanisme de déplacement de l'ovale (45 mn)
- Acquisition de compétences en Swing (60 mn)
- Documentation du projet (60 mn)

Séance 2

- Liste des tâches :
- Analyse du problème (15 mn)
- Conception, développement et test du mécanisme de utilisation des threads pour descendre la hauteur d'un ovale (45 mn)
- Conception, développement et test: dessiner un circuit sous la forme d'une ligne brisée autour de laquelle l'ovale doit se promener (90 mn).
- Documentation du projet (60 mn)

Séance 3

- Liste des tâches :
- Analyse du problème (15 mn)
- Conception, développement et test: détection des collisions vérifier lors de chaque changement de hauteur si l'ovale est sorti de la ligne brisée(60 mn)
- Conception, développement et test: ajouter des oiseaux qui se déplacent en battant des ailes dans le décors(90 mn)
- Résoudre sur problème: une surcharge au niveau de l'affichage, chaque thread alerte le JPanel (20mn)
- Améliorer la courbe brisée
- Rendre les mouvements plus dynamiques
- Créer un .jar exécutable
- Calculez un bruit de Perlin en deux dimensions pour dessiner des ovales représentant des nuages avec une structure(60mn)
- Documentation du projet (45 mn)

Voici un diagramme de Gantt qui décrit mon planification de temps pour l'avancement du projet et la répartition des tâches. Le projet dans son ensemble sera avancé sur cette base.

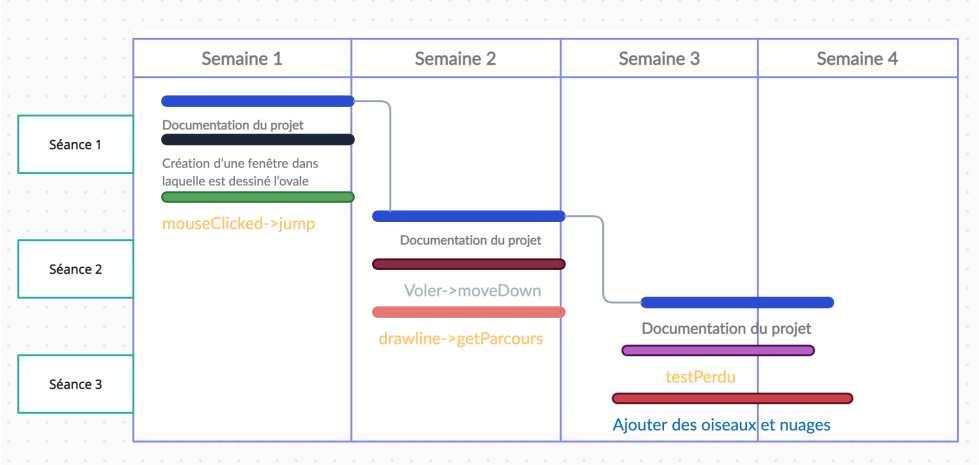

Figure 2 – Diagramme de Gantt de ces activités au cours de la séance

4. Conception générale

Pour la conception générale, j'utilise le modèle MVC (Modèle-Vue-Contrôleur). Le MVC sépare les données de l'application (Modèle : Oiseau, Etat, Parcours), la manière dont ces données sont présentées à l'utilisateur (Vue : Affichage, Fenêtre, PerlinNoise, VueOiseau) et la logique de contrôle qui les relie (Contrôle : Avancer, Control, Voler).

C'est une bonne solution car elle offre une meilleure organisation du code et une plus grande flexibilité dans le développement de l'application. En séparant les données, la présentation et la logique, je peux facilement apporter des modifications à l'une de ces parties sans affecter les autres. De plus, cela permet une plus grande simplicité dans le développement en solitaire, car je peux travailler sur une partie distincte du modèle. Enfin, cela peut également améliorer les performances de l'application car certaines parties peuvent être optimisées sans affecter les autres. Ainsi mes packages sont les suivants :

- **Model: (Bird, Etat, Parcours)**
- **Control: (Avancer, Control, Voler)**
- **View: (Affichage, Fenetre, PerlinNoise, VueOiseau)**

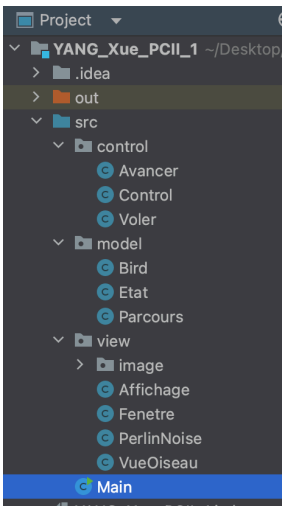

Figure 3 – Arborescence de mon projet

5. Conception détaillée

La structure de diagramme de classe pour les trois packages de mon modèle MVC (Modèle-Vue-Contrôleur) est la suivante :

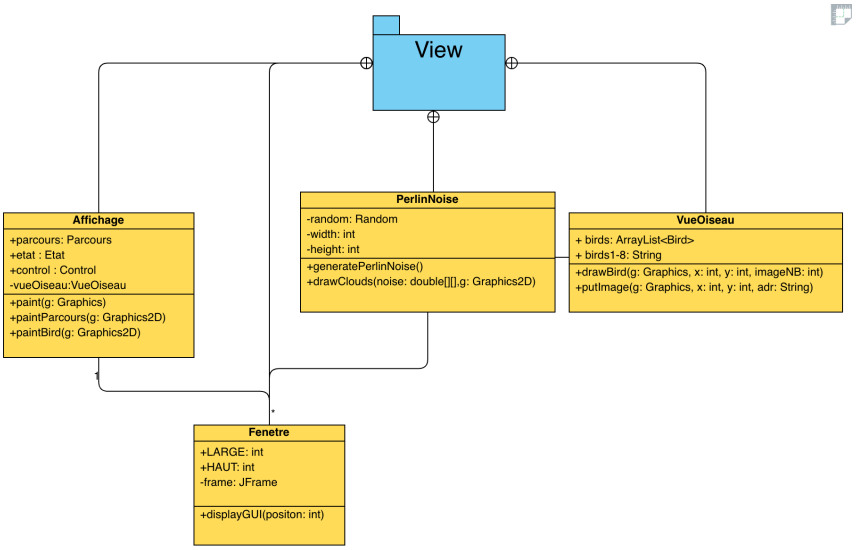

Figure 4 – Diagramme UML du package View

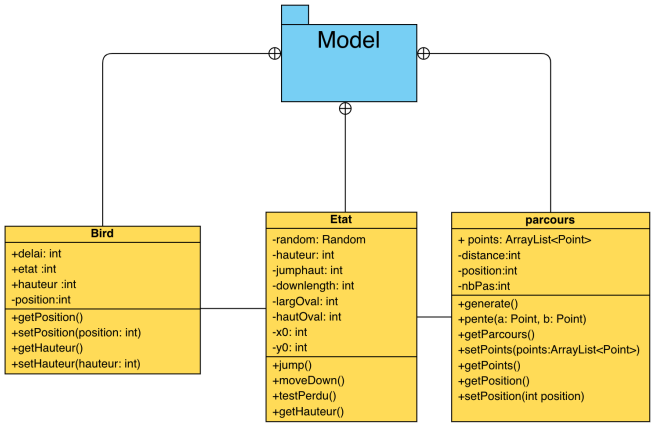

Figure 5 – Diagramme UML du package Model

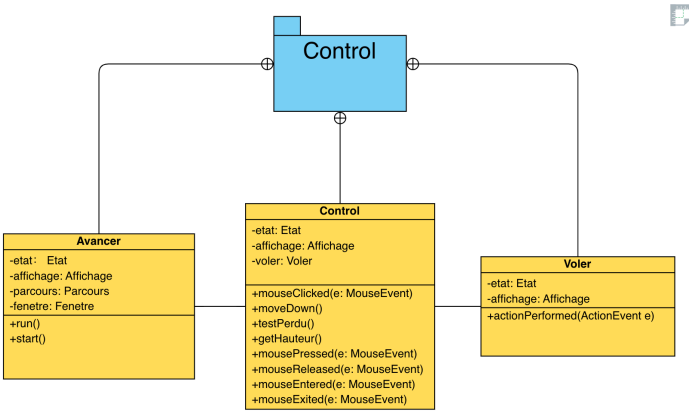

Figure 6 – Diagramme UML du package Control

Pour la fenêtre ovale, j'ai implémenté l'API Swing et utilisé la classe JPanel pour définir ses dimensions et les constantes associées.

```java
private JFrame frame;
/**
 * Constructeur de la fenêtre.
 * @param title   le titre de la fenêtre
 * @param affichage  le composant graphique à afficher dans la fenêtre
 */
1 usage
public Fenetre(String title, Affichage affichage) {
    /**JFrame: implémenter une fenêtre*/
    JFrame frame = new JFrame();
    this.frame = frame;

    /**ajouter le composant graphique*/
    frame.add(affichage);
    frame.setTitle(title);

    /**Définir ses dimensions (largeur et hauteur) */
    frame.setMaximumSize(new Dimension(LARGE, HAUT));
    frame.setMinimumSize(new Dimension(LARGE, HAUT));
    frame.setLocationRelativeTo(null);
    frame.setResizable(false);
    /**assembler et rendrer visible les composants*/
    frame.pack();
    frame.setVisible(true);
    /**fermer la fenêtre*/
    frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
```

```java
/** Méthode paint qui est appelée par le système d'exploitation à chaque mise à jour de l'affichage.
 * @param g instance de la classe java.awt.Graphics
 */
public void paint(Graphics g) {
    super.paint(g);
    /**dessiner un ovale bleu*/
    g.setColor(Color.blue);
    g.drawOval(parcours.points.get(0).x, etat.getHauteur(), etat.largOval, etat.hautOval);
    /**dessiner les lignes noirs*/
    g.setColor(Color.black);
    paintParcours((Graphics2D) g);
    /**dessiner l'oiseau*/
    try {
        paintBird((Graphics2D) g);
        paintBird((Graphics2D) g);
    } catch (IOException | InterruptedException e) {
        throw new RuntimeException(e);
    }
    this.perlinNoise.drawClouds(this.perlinNoise.generatePerlinNoise(), (Graphics2D) g);
}
```

J'ai également créé une classe nommée "Etat" qui gère les données du jeu concernant l'ellipse, et les méthodes "jump" et "moveDown" pour contrôler sa montée et sa descente. Dans la classe de contrôle, j'ai ajouté une méthode "mouseClicked" qui permet à l'utilisateur de contrôler la montée de l'ellipse en cliquant avec la souris.

```java
public Etat() {
    Parcours parcours1 = new Parcours();
    this.parcours=parcours1;
    this.hauteur = parcours.points.get(1).y-hautOval/2;
    /**centreOvale (x0,y0)*/
    x0 =parcours.points.get(0).x+20;
    y0 = this.hauteur+hautOval/2;
    this.exit = testPerdu();
}
/**Permet de faire sauter l'ovale en augmentant sa hauteur.* */
1 usage
public void jump() {
    if (this.hauteur <= 0){
        this.hauteur= 0;
    }else{
        this.hauteur -= jumphaut;
    }
}
/**Permet de faire descendre l'ovale en modifiant sa hauteur.
 * sans sortir de la zone de dessin*/
1 usage
public void moveDown(){
    if (this.hauteur + hautOval + downlength <= Fenetre.HAUT ) {
        this.hauteur += downlength;
    }
}
```

```java
/** Constructeur */
2 usages
public Control(Etat etat, Affichage affichage) {
    this.etat = etat;
    this.affichage = affichage;
}
/**
 *  mouseClicked est appelée chaque fois que l'utilisateur clique dans la zone d'affichage
 *  @param e est une intance de la classe java.awt.MouseEvent
 * */
@Override
public void mouseClicked(MouseEvent e) {
    if (!etat.exit){
        etat.exit = etat.testPerdu();
        //System.out.println("control");
        etat.jump();
        affichage.repaint();
    }

}
```

J'ai ajouté une classe "Voler" au package de contrôle pour contrôler la descente automatique de l'ellipse à l'aide de threads multiples.

```java
public Voler(Etat etat, Affichage affichage) {
    this.etat = etat;
    this.affichage = affichage;
    initTimer();
}
1 usage
private void initTimer() {
    Timer timer = new Timer( delay: 400, new ActionListener() {
        @Override
        public void actionPerformed(ActionEvent e) {
            if (!etat.exit) {
                etat.exit = etat.testPerdu();
                etat.moveDown();
                /** forcer le dessin */
                affichage.revalidate();
                affichage.repaint();
            } else {
                ((Timer) e.getSource()).stop();
            }
        }
    });
    timer.start();
}
```

Pour générer des lignes qui répondent aux exigences de la pente, j'ai créé une classe "Parcours". La méthode "pente" calcule la pente et la position des points, la méthode "generate" génère aléatoirement les points nécessaires et la méthode "getParcours" enlève les points sortis de l'écran et ajoute les nouveaux points.

```java
public Parcours() {
    position = 0;
    points = new ArrayList<>();
    generate();
}
/**
 * Génère les points nécessaires pour le parcours
 */
1 usage
public void generate() {
    Random rand = new Random();
    int currentIndex = 0;
    for (int i = 0; i <= nbPas; i++) {
        Point p = new Point();
        p.x = (Fenetre.LARGE) / nbPas * (i);
        p.y = rand.nextInt(Fenetre.HAUT);
        if (points.size() == 0) {
            points.add(p);
            currentIndex++;
        } else {
            while (Math.abs(pente(p, points.get(currentIndex - 1))) > 0.5) {
                p.y = rand.nextInt( bound: Fenetre.HAUT-50);
            }
            points.add(p);
            currentIndex++;
        }
```

```java
/**
 * Calcule la pente entre deux points
 *
 * @param a le premier point
 * @param b le deuxieme point
 * @return la pente entre a et b
 */
3 usages
public float pente(Point a, Point b) { return (float) ((b.y)- (a.y)) / (b.x - a.x); }
```

```java
/** Renvoie les points visibles du parcours
 *
 *  @return ArrayList<Point>   les points visibles
 */
1 usage
public ArrayList<Point> getParcours() {
    /** Déplace les points */
    for (Point tmp : points) {
        tmp.x -= distance;
    }
    Random rand = new Random();
    /** Retire le premier point qui est sorti de la fenêtre */
    points.remove( index: 0);
    /** Retire le premier point qui est sorti de la fenêtre */
    int size = points.size();
    Point newP = new Point();
    newP.x = points.get(size - 1).x + distance;
    newP.y = rand.nextInt( bound: Fenetre.HAUT-50);

    Point lastPoint = points.get(size - 1);
    while (Math.abs(pente(newP, lastPoint)) > 0.5) {
        newP.y = rand.nextInt( bound: Fenetre.HAUT-50);
    }
        points.add(newP);
    return points;
}
```

La classe "Affichage" comprend la méthode "paintParcours" pour dessiner les lignes requises.

```java
/** connecter tous les points de la Liste
 *  @param g est une intance de la classe java.awt.Graphics
 */
1 usage
public void paintParcours(Graphics2D g){

    for (int i = 0; i < this.etat.parcours.points.size() - 1; i++) {
        Point p1 = this.etat.parcours.points.get(i);
        Point p2 = this.etat.parcours.points.get(i+1);
        g.drawLine(p1.x, p1.y, p2.x, p2.y);

    }
}
```

J'ai écrit une classe "Avancer" dans le package de contrôle pour déplacer en continu les lignes de parcours aléatoires. J'utilise des threads et la méthode "thread.sleep" pour redessiner toutes les 800 millisecondes et calcule le score du joueur à chaque déplacement réussi.

```java
public Avancer(Etat e, Parcours parcours, Affichage affichage,Fenetre fenetre){
    this.parcours = parcours;
    this.etat=e;
    this.affichage = affichage;
    this.fenetre=fenetre;
}
/**  Vérifiez si l'ovale est hors ligne, sinon, déplacez les lignes
 * */
@Override
public void run() {
        while(!etat.exit){
            try{
                etat.parcours.getParcours();
                etat.parcours.setPosition(etat.parcours.getPosition() + 1);
                System.out.println(etat.parcours.getPosition());
                etat.exit = etat.testPerdu();
                Thread.sleep( millis: 800);
            }catch (InterruptedException e){
                e.printStackTrace();
            }
            /**forcer le dessin*/
            this.affichage.revalidate();
            this.affichage.repaint();
            if (etat.exit){
                fenetre.displayGUI(etat.parcours.getPosition());
            }
```

La méthode "testPertu" de la classe Etat vérifie si l'ellipse quitte la voie de déplacement et, dès qu'elle quitte, arrête immédiatement tous les déplacements ainsi que l'affichage d'une boîte de message indiquant le score du joueur.

```java
/**
 * tester si l'ovale est sorti de la ligne brisée
 *
 * @return vrai si l'ovale est sorti de la ligne brisée
 */
4 usages
public boolean testPerdu(){
    Point p1 = parcours.points.get(0);
    Point p2 = parcours.points.get(1);
    float pente = parcours.pente(p1,p2);
    x0 =parcours.points.get(0).x+20;
    y0 = this.hauteur+hautOval/2;
    float y =  pente*(x0)+((p1.y)-pente*p1.x);
    if (y0 > y + 100||y0 < y - 100){
        return true;
    }else {
        return false;
    }
}
```

```java
/**
 * Afficher un message avec le score de l'utilisateur.
 *
 * @param positon     le score du joueur à la fin de le jeu
 */
1 usage
public void displayGUI(int positon) {
    JOptionPane.showMessageDialog(this.frame,
            message: "" + positon,
            title: "Your position is",
            JOptionPane.INFORMATION_MESSAGE);
}
```

J'ai ajouté une classe "Bird" au package "Model" et une classe "VueOiseau" au package "View" pour charger l'image de l'oiseau et pour animer son mouvement en changeant son image et sa position à un certain taux.

```java
public class Bird {
    /**indique le temps (en millisecondes) entre chaque mise à jour de l'affichage pour l'oiseau*/
    1 usage
    public static int delai;

    /**permet de savoir dans quelle position est l'oiseau*/
    1 usage
    private int etat;
    /**définit la hauteur de l'oiseau dans la fenêtre graphique */
    3 usages
    private int hauteur;

    /**définit l'abscisse de l'oiseau*/
    4 usages
    private int position;

    1 usage
    public Bird() {
        Random rand = new Random();
        this.delai = rand.nextInt( bound: 50);
        this.hauteur = rand.nextInt(Fenetre.HAUT);
        this.position = rand.nextInt(Fenetre.LARGE);;
        this.etat=this.position;
    }
```

```java
public void VueOiseau(){

}
/**
 * Dessinez l'image de l'oiseau dans le programme en fonction du numéro de série
 *
 * @param g      le contexte graphique
 * @param x      l'obscisse de l'image
 * @param y      la coordonnée verticale de l'image
 * @param imageNB  le numéro de l'image
 */
2 usages
public  static void drawBird(Graphics g, int x, int y, int imageNB){

    switch (imageNB){
        case 1: putImage(g,x,y,bird1);
        case 2: putImage(g,x,y,bird2);
        case 3: putImage(g,x,y,bird3);
        case 4: putImage(g,x,y,bird4);
        case 5: putImage(g,x,y,bird5);
        case 6: putImage(g,x,y,bird6);
        case 7: putImage(g,x,y,bird7);
        case 8: putImage(g,x,y,bird8);
    }
}
```

```java
private static void putImage(Graphics g, int x, int y, String adr) {
    ImageIcon icon = new ImageIcon(adr);
    Image image = icon.getImage();
    g.drawImage(image, x, y,  width: 100, height: 100, observer: null );
}
```

Enfin, j'ai ajouté une classe "PerlinNoise" au package "View" pour générer du bruit de Perlin et dessiner un effet de fond nuageux dynamique.

```java
public PerlinNoise(int width, int height) {
    this.width = width;
    this.height = height;
    this.random = new Random();
}
/**
 * Génère un bruit de Perlin
 *
 * @return une matrice de doubles représentant le bruit de Perlin
 */
1 usage
public double[][] generatePerlinNoise() {
    double[][] noise = new double[width][height];
    // Initialiser les valeurs de bruit
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            noise[x][y] = random.nextDouble()* 0.5;
        }
    }
    return noise;
}
```

```java
public void drawClouds(double[][] noise,Graphics2D g) {
    int count = 0;
    // Dessiner des ovales uniquement tous les 5 pixels en x et en y
    for (int x = 0; x < width; x++) {
        for (int y = 0; y < height; y++) {
            double n = noise[x][y];
            int ovalSize = (int) (n * 50);
            if (x % 5 == 0 && y % 5 == 0) {
                // Dessiner seulement 1/10 des ovales
                if (count++ < (width * height) / 10) {
                    int ovalX = x * ovalSize;
                    int ovalY = y * ovalSize;
                    int ovalWidth = ovalSize;
                    int ovalHeight = ovalSize;
                    g.setColor(new Color( r: 240,  g: 255,  b: 255));
                    g.fillOval(ovalX, ovalY, ovalWidth, ovalHeight);
                } else {
                    break;
                }
            }
        }
        if (count >= (width * height) / 10) {
            break;
        }
    }
```

6. Documentation utilisateur

Prérequis : Java avec un IDE et Makefile

Mode d'emploi (cas IDE) : Importez le projet dans votre IDE, sélectionnez la classe Main à la racine du projet puis « Run as Java Application ». Cliquez sur la fenêtre pour faire monter l'ovale.

Mode d'emploi (cas Makefile) : Ouvrez le terminal dans le chemin du dossier, entrez "make" dans le terminal et appuyez sur Entrée.

7. Documentation développeur

Dans le but d'améliorer la qualité de la courbe brisée, nous utiliserons des épaisseurs et des couleurs différentes pour créer de la consistance. Cela donnera à la courbe brisée un aspect plus uniforme et cohérent. De plus, nous utiliserons des courbes de Bézier pour arrondir les angles, ce qui donnera à la courbe brisée un aspect plus doux et naturel, ainsi qu'une apparence plus esthétique pour l'utilisateur final. Les courbes de Bézier permettent de contrôler la forme de la courbe en utilisant des points de contrôle définis par l'utilisateur, ce qui permet de créer des courbes plus complexes et réalistes tout en conservant une certaine flexibilité pour adapter la forme de la courbe aux besoins de l'application. En conclusion, l'utilisation de différentes épaisseurs et couleurs, ainsi que l'utilisation de courbes de Bézier, amélioreront la qualité de la courbe brisée et offriront une meilleure expérience utilisateur et une présentation visuelle plus attrayante pour les utilisateurs.

8. Conclusion et perspectives

Au cours de ce projet, j'ai eu des défis dans la compréhension et la construction du modèle MVC. Cependant, grâce à ma propre recherche d'informations et aux explications de mon professeur, j'ai finalement pu mieux comprendre ce modèle.

Le générateur de points avec une pente conforme aux exigences et le dessin en tant que ligne mobile a été un défi pour moi au départ, mais après de nombreuses tentatives et modifications de code, j'ai finalement réussi à écrire un code qui respecte les exigences.

Enfin, j'ai également essayé les sujets facultatifs subséquents, une courbe de Bézier, mais je n'ai pas encore réussi à les intégrer. Si j'ai encore du temps, j'espère pouvoir les terminer pour compléter ce projet.

En résumé, ce projet a été une expérience enrichissante pour moi en termes de développement de mes compétences en programmation et en compréhension de différents modèles de conception. J'ai hâte de continuer à apprendre et à développer mes compétences en programmation.