DOCUMENTATION TECHNIQUE – POND SIMULATOR

Description du projet :

Ce projet présente une simulation de mare, les canards doivent manger des nénuphars pour rester en vie et grandir, s'ils ne mangent pas à temps ils disparaissent, le dernier canard vivant devient alors invincible.

I.Classes et interfaces

1. IPondEntity

Les interfaces servent à décrire un comportement pour un ensemble de classes, dans le cas du Pond Simulator, l'interface définit donc le comportement des entités « duck », « lilypad » et « rock ».

```
public interface IPondEntity {
    public void update();
    public void render(Graphics2D g);

    public Vector2D getPosition();
    public Vector2D getSize();
}
```

• 2 fonctions void :

- update() → actualise les positions et comportement de objets des classes
- render() → gère l'affichage des éléments de la mare de Pond Simulator.

• 2 fonctions dites « getter » qui retournent la position et la taille des entités.

2. Classe duck

```
public Duck(int x, int y) { //nouvelle instance Duck en x, y; avec les caractéristiques suivantes
    this.x = x;
    this.y = y;
    this.width = this.height = 50;
    this.level = 1;
    this.rotation = 45;
    this.image = PondManager.getSingleton().getDuckImg1();
    this.remainingTime = 1000;
}
```

```
public void update() { //fonction décrite par l'interface
    this.forward();
    this.remainingTime -=1;
}
```

• this.forward() est une fonction void qui gère le déplacement de chaque canard.

```
public void forward() {     //déplacement selon position, angle et vitesse données
    Double translated = MathUtils.translate2D(this.x, this.y, this.rotation, getMoveSpeed(this.level));
    this.x = translated.x;
    this.y = translated.y;

}
```

• Cette fonction permet de traduire en vecteur le déplacement de chaque canard, selon la vitesse du canard définie par son niveau et l'angle.

```java
public void levelUp() { //augmentation du niveau
    if (level >= 10) return;

    level += 1;

    this.width = this.height += 5;

    if (level == 10) { // Becomes chief
        this.image = PondManager.getSingleton().getDuckImg2();
    }
}
```

• La taille des canards varie selon leur niveau (chaque nénuphar mangé augmente le niveau de 1 point), arrivés au niveau 10 l'image du roi des canards est récupérée par un singleton sur la fonction responsable de récupérer l'image correspondante.

```java
int getMoveSpeed(int level) {   //vitesse selon niveau du canard
    if (level < 10){
        return 2;
    } else {
        return 1;
    }
}
```

• Retourne la vitesse selon le viveau du canard

Paramètre : le niveau du canard

Return : la vitesse

```java
public void render(Graphics2D g) {      //rendu graphique du canard en cours
    PondManager pm = PondManager.getSingleton();
    BufferedImage duckImg = null;
    Color color;
    if (level <= 3) { // bébé canard
        color = new Color( r: 255, g: 255, b: 0);
        duckImg = pm.getDuckImg1();
    }
    else if (level < 10) { // canard
        color = new Color( r: 255, g: 255, b: 0);
        duckImg = pm.getDuckImg1();
    }
    else { // chef de la tribu des canards
        color = new Color( r: 220, g: 220, b: 220);
        duckImg = pm.getDuckImg2();
    }

    g.drawImage(GraphicUtils.rotateImageByDegrees(duckImg, rotation), (int)this.x, (int)this.y, width, height,  observer: null);
}
```

• Il s'agit d'une fonction décrite par l'interface, selon le niveau, la rotation, la taille et la position du canard elle retourne et dessine l'image correspondante à chaque canard.

Paramètre : Graphics g est le rendu graphique du plateau.

```java
@Override
public Vector2D getPosition() { //getter sur la position du canard, return un vector2D
    return new Vector2D(this.x, this.y);
}

@Override
public Vector2D getSize() {     //getter de dimensions, return un vector2D
    return new Vector2D(this.width, this.height);
}
```

• Il s'agit de 2 fonctions « getter » pour récupérer la taille et la position des canards en temps réel.

Return : respectivement position et taille

```java
public void setRotation(double rot, boolean spamProtect) {  //calcule le temp de la dernière rotation, afin de limiter les glitch
    long nowTimeMilli = System.currentTimeMillis();
    if (spamProtect && nowTimeMilli - this.lastChangedRot < 100) return;

    if (rot < 0) rot += 360;
    this.rotation = rot;
    lastChangedRot = nowTimeMilli;
}
```

• Cette fonction définie la prochaine rotation à exécuter, spamProtect limite les glitch, afin qu'il y ait un temp limite avant la prochaine rotation.

```
public double getRotation() { //getter de la prochaine rotation
    return this.rotation;

}
```

• C'est une fonction « getter », qui récupère la rotation actuelle en degrés.

Return : rotation

3. Lilypad

```
public class Lilypad implements IPondEntity {
    private int x;
    private int y;
    private int width;
    private int height;

    public Lilypad(int x, int y) {
        this.x = x;
        this.y = y;
        this.width = 70;
        this.height = 70;
    }
```

• Lilypad est une classe définie par l'interface IPondEntity, avec les caractéristiques décrite ci-contre.

```java
@Override
public void update() {

}

@Override
public void render(Graphics2D g) {

    PondManager pm = PondManager.getSingleton();
    Image lilypadImg = null;
    lilypadImg = pm.getLilypadImg();
    g.drawImage(lilypadImg, this.x, this.y, this.width, this.height,  observer: null);

}
```

• update() et render() sont des comportements décrit par l'interface IPondEntity :

Render() dessine les nénuphars en (x, y).

```java
@Override
public Vector2D getPosition() {
    return new Vector2D(this.x, this.y);
}

@Override
public Vector2D getSize() {
    return new Vector2D(this.width, this.height);
}
```

• Ce sont 2 « getter » qui permettent de récupérer la taille et la position des nénuphars.

---

3. Rock

• Rock est aussi une classe définie par l'interface IPondEntity avec les paramètres suivants.

```java
public class Rock implements IPondEntity {
    private int x;
    private int y;
    private int width;
    private int height;
    private Image image;

    public Rock(int x, int y) {
        this.x = x;
        this.y = y;
        this.width = 110;
        this.height = 110;
    }
```

```java
@Override
public void update() {
}

@Override
public void render(Graphics2D g) {
    PondManager pm = PondManager.getSingleton();
    Image RockImg = null;
    RockImg = pm.getRockImg();
    g.drawImage(RockImg, this.x, this.y, this.width, this.height,  observer: null);
}
```

• les fonctions définies par l'interface permettent l'affichage des pierres en position x, y.

Paramètre : Graphics2D

```
    @Override
    public Vector2D getPosition() { return new Vector2D(this.x, this.y); }

    @Override
    public Vector2D getSize() { return new Vector2D(this.width, this.height); }
}
```

• Les 2 fonctions de « getter » pour récupérer la taille et la position des pierres.

Return : vecteurs à la position des pierres.

II. Pond Manager

```
private ArrayList<IPondEntity> entities = new ArrayList<>();
```

• Cette liste de l'interface regroupe toutes les entités de la mare (canards, pierres, nénuphars..)

```
public static PondManager getSingleton() {
    if (_instance == null) {
        _instance = new PondManager();
    }
    return _instance;
}
```

• L'instance est créée grâce à un singleton, ce qui va assurer la création unique d'instance.

Return : la nouvelle instance de pondManager

```java
public void loadAssets() throws IOException {
    this.rockImg = ImageIO.read(new File( pathname: "assets/rock.png"));
    this.duckImg1 = ImageIO.read(new File( pathname: "assets/duck1_right.png"));
    this.duckImg2 = ImageIO.read(new File( pathname: "assets/duck2_right.png"));
    this.lilypadImg = ImageIO.read(new File( pathname: "assets/lilypad.png"));

}
```

• Récupération des images nécessaires à l'affichage graphique des objets.

```java
public static class SoundAnimation {
    public static synchronized void play(final String fileName)
    {
        new Thread(new Runnable() {
            public void run() {
                try {
                    Clip sound = AudioSystem.getClip();
                    AudioInputStream inputStream = AudioSystem.getAudioInputStream(new File(fileName));
                    sound.open(inputStream);
                    sound.start();
                } catch (Exception e) {
                    System.out.println("SoundAnimation: error " + e.getMessage() + " for " + fileName);
                }
            }
        }).start();

    }
}
```

• Cette fonction permet l'exécution de sons grâce à l'instructions SoundAnimation.play(« path/toFile/sound.wav ») par exemple.

• Chaque son est défini sur un nouveau thread.

```
    public void spawnDuck() {
    int randx = ThreadLocalRandom.current().nextInt( origin: 0,  bound: 1280);
    int randy = ThreadLocalRandom.current().nextInt( origin: 0,  bound: 800);
    entities.add(new Duck(randx, randy));
}

public void spawnLilypad() {
    int randx = ThreadLocalRandom.current().nextInt( origin: 0,  bound: 1280);
    int randy = ThreadLocalRandom.current().nextInt( origin: 0,  bound: 800);
    entities.add(new Lilypad(randx, randy));

}

public void spawnRock() {
    int randx = ThreadLocalRandom.current().nextInt( origin: 0,  bound: 1280);
    int randy = ThreadLocalRandom.current().nextInt( origin: 0,  bound: 800);
    entities.add(new Rock (randx, randy));

}
```

• Ces 3 fonctions permettent de faire apparaître un canard, un nénuphar ou une pierre à un endroit aléatoire du plateau.

• La génération aléatoire est gérée par un thread.

La fonction update() du PonManager :

Dans un premier temps, on exécute pour chaque élément définit par l'interface la fonction update().

```
    for (IPondEntity entity : entities) {
        entity.update();
    }
```

Ensuite, on définit une nouvelle ArrayList dépendante de l'interface IPondEntity, cette liste toRemove désignera les éléments à supprimer de la liste des objets présents.

```
ArrayList<IPondEntity> toRemove = new ArrayList<>();
```

```java
for (IPondEntity entity : entities) {
    if (!(entity instanceof Duck)) continue;
    Duck duck = (Duck) entity;
    Vector2D pos = duck.getPosition();
    Vector2D size = duck.getSize();
    double rotation = duck.getRotation();
    double angle = Math.toRadians(rotation);
```

• Pour chaque entité qui n’est pas un canard, ignorer la suite,

• Cependant, tout ce qui est un canard (duck) récupèrera sa position actuelle, sa taille et la rotation suivante.

```java
// pond borders
double newRot = 0;
boolean shouldUpdateRot = false;
if (pos.x <= 0) { // LEFT
    Vector2D vec = MathUtils.reflect(new Vector2D( x: 1,  y: 0).rotate(-angle), new Vector2D( x: 1,  y: 0).rotate(Math.toRadians(180)));
    newRot = vec.getRotation();
    shouldUpdateRot = true;
} else if (pos.x + size.x >= width) {  // RIGHT
    Vector2D vec = MathUtils.reflect(new Vector2D( x: 1,  y: 0).rotate(-angle), new Vector2D( x: 1,  y: 0).rotate(Math.toRadians(0)));
    newRot = vec.getRotation();
    shouldUpdateRot = true;
}

if (pos.y <= 0) { // TOP
    Vector2D vec = MathUtils.reflect(new Vector2D( x: 1,  y: 0).rotate(-angle), new Vector2D( x: 1,  y: 0).rotate(Math.toRadians(270)));
    newRot = vec.getRotation();
    shouldUpdateRot = true;
} else if (pos.y + size.y >= height) { // BOTTOM
    Vector2D vec = MathUtils.reflect(new Vector2D( x: 1,  y: 0).rotate(-angle), new Vector2D( x: 1,  y: 0).rotate(Math.toRadians(90)));
    newRot = vec.getRotation();
    shouldUpdateRot = true;
}
```

• Cette partie de la fonction permet de définir les limites de la mare, grâce à des vecteurs, si un canard franchi ces vecteurs alors il tournera avec un angle de ± 90 ° de + qu’auparavant.

```
//other entities
for (IPondEntity entity2 : entities) {
    if (entity2 == entity) continue;
    boolean isColliding = false;
    Vector2D pos2 = entity2.getPosition();
    Vector2D size2 = entity2.getSize();

    // Apply hitbox tolerance
    if (entity2 instanceof Duck) {
        pos2.x += tolerance;
        pos2.y += tolerance;
        size2.x -= tolerance*2;
        size2.y -= tolerance*2;
        if (((Duck) entity2).remainingTime < 0) {
            toRemove.add(entity2);
        }
    }


    if (pos.x + size.x > pos2.x &&
        pos.x < pos2.x + size2.x &&
        pos.y + size.y > pos2.y &&
        pos.y < pos2.y + size2.y) {
        isColliding = true;
    }
```

• Pour chaque entité de la mare, si l'entité testée est du même type que celle en cours, ignorer la suite.

• Si l'entité est un canard, lui ajouter une marge de tolérance, cette tolérance permet de gérer la hitbox de chaque élément selon sa taille (ratio).

• Si le temps restant remainingTime de l'entité (canard) est inférieure à 0, ajouter cet élément à la liste des éléments à supprimer.

• Si la position de l'entité en cours est à la même position qu'une autre entité, alors signaler qu'il y a une collision.

```
if (entity2 instanceof Lilypad && isColliding) {

    duck.levelUp();
    SoundAnimation.play( fileName: "assets/Honk.wav");
    toRemove.add(entity2);
    nbLilypads -=1;
    duck.remainingTime += 200;
```

• Si l'entité est un nénuphar, augmenter le niveau, jouer un son d'animation, et supprimer le nénuphar visé.

```
} else if (entity2 instanceof Rock && isColliding) {
  int angleNormal = 0;
  if (pos.x <= pos2.x) angleNormal = 0;
  else if (pos.x + size.x >= pos2.x + size2.x) angleNormal = 180;
  else if (pos.y <= pos2.y) angleNormal = 90;
  else if (pos.y + size2.y >= pos2.y + size2.y) angleNormal = 270;

  Vector2D vec = MathUtils.reflect(new Vector2D( x: 1,  y: 0).rotate(-angle), new Vector2D( x: 1,  y: 0).rotate(Math.toRadians(angleNormal)));
  newRot = vec.getRotation();
  shouldUpdateRot = true;
}
```

• Si l'entité est un caillou et entre en collision, faire tourner le canard.

```
    entities.removeAll(toRemove);
```

• Supprime toutes les entités de la liste à supprimer.

```
if (nbDucks < 10) {
    spawnDuck();
    nbDucks += 1;
}
if (nbRocks < 5) {
    spawnRock();
    nbRocks += 1;
}
if (nbLilypads <= 15) {
    spawnLilypad();
    nbLilypads += 1;
}
```

• Permet de gérer le nombre d'entité de chaque classe.

```
public void render(Graphics2D g) {
    // entities.forEach(e -> e.render(g));

    for (IPondEntity entity : entities) {
        entity.render(g);
        int tolerance = 15;
        Vector2D pos = entity.getPosition();
        Vector2D size = entity.getSize();

        if (entity instanceof Duck) {
            pos.x += tolerance;
            pos.y += tolerance;
            size.x -= tolerance*2;
            size.y -= tolerance*2;
        }

    }
}
```

• Permet d'appeler la fonction render de chaque classe définie par l'interface, afin de dessiner chaque élément de la liste de entités.

```java
public BufferedImage getRockImg() {
    return rockImg;
}

public BufferedImage getDuckImg1() {
    return duckImg1;
}

public BufferedImage getDuckImg2() {
    return duckImg2;
}

public BufferedImage getLilypadImg() {
    return lilypadImg;
}
```

• Fonctions permettant de récupérer les images attribuées à chaque élément.

III. Outils

Pour ce projet, 3 fichiers d'outil permettent le fonctionnement des différentes fonctions du programme principal :

- GraphicUtils : Permet de gérer l'affiche du plateau ainsi que toutes les entités (Graphics2D),
- MathUtils : Permet de gérer les déplacements ainsi que les angles de rotation des canards,
- Vector2D : Permet de traduire les positions en vecteurs, ainsi que les déplacements des canards.

IV. Gestion et paramètres de la fenêtre de jeu

Les 3 fichiers restants assurent le lancement du jeu :

- GameLauncher : crée et instancie la fenêtre de jeu,
- GamePanel : Définit les dimensions de la fenêtre, les caractéristiques générales (anti-aliasing, render quality, …), gère les messages d'erreur de chargement, gère le timer général du jeu (rythme de jeu), et définie la couleur de l'arrière-plan.
- Window : Définit le titre de la fenêtre et créée une nouvelle fenêtre.

VI. Diagramme de Classe