Dossier de projet Final

Session 2019

Death Run « Pokémon »

Figure 1: Zone de jeu

Table des matières

1/ Présentation générale

1.1/ Présentation de l'équipe

Les élèves inclus dans le projet actuel sont :

- MOUCHAGUE Dorian : Chef de projet non-objet
- SARRAUD Clément : Programmeur Objet
- ROUGIER Clément : Programmeur Objet
- GUILLOT Axel : Programmeur Objet

1.2/ Présentation du projet

Les 3 sujets concernant l'Intelligence Artificielle de cette année sont :

- Système Expert
- Logique floue
- Algorithme génétique

Notre choix s'est porté sur l'algorithme génétique car c'est le sujet qui était le plus intéressant à nos yeux, et nous avons donc décider de concevoir un « Death Run » appelé aussi une Course de la mort.

Le projet consiste en la génération d'individus devant traverser verticalement la zone de jeu en évitant les pièges jusqu'à atteindre l'arrivée. Le but est donc de voir les différentes générations apprendre de leurs échecs jusqu'à arriver tout en bas de l'écran après un nombre « n » de génération.

2/ Présentation technique

2.1/ Création de l'environnement

La première chose que l'on a eu à faire pour ce projet après avoir défini notre sujet et ses composantes a été de créer l'interface de notre jeu.

L'écran du jeu a été séparé en 2 panels, celui de gauche servant simplement à l'affichage de l'écran de jeu et le panel de droite servant à différentes choses :

- Informations sur l'avancée du programme (nombre d'individus vivants et de leur génération)
- Le paramétrage du jeu (nombre d'individus, de bombes et de génération)
- Le bouton pour lancer le programme et le relancer quand il est fini
- Ajout du personnage le plus proche du but en haut à droite

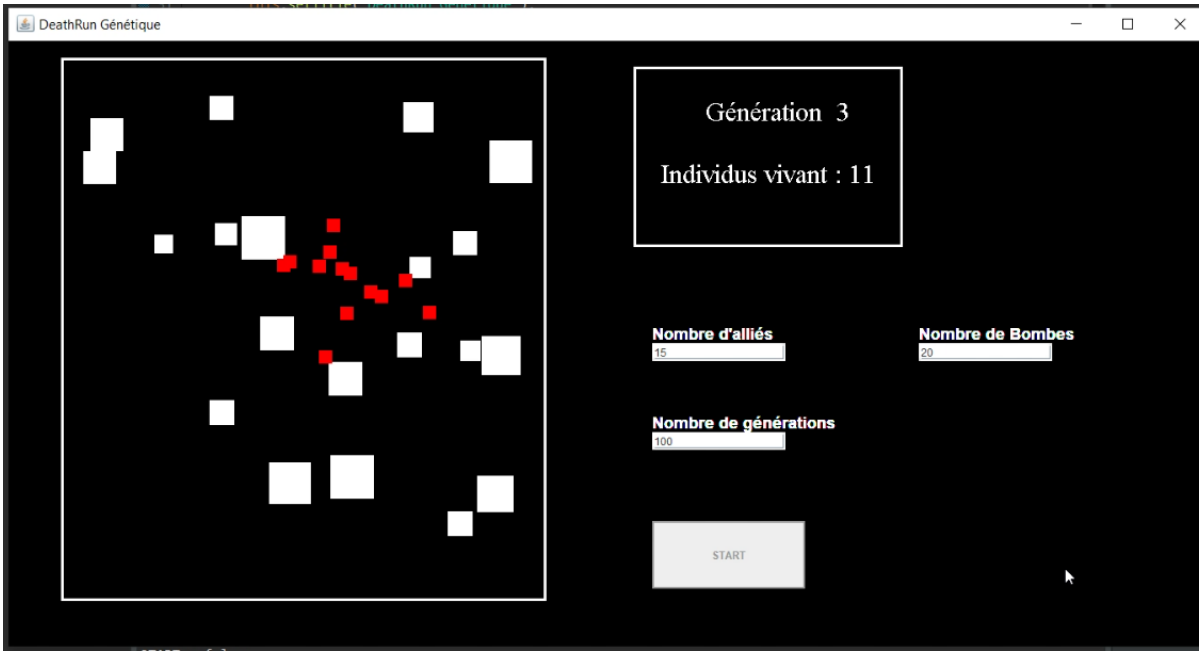

Figure 2: Jeu en cours de développement

2.2/ Création des personnages et pièges

Une fois l'interface créée, il a fallu mettre en place les alliés et les bombes, pour cela rien de compliqué. Au début on s'est contenté via une méthode (Graphics) permettant de dessiner des formes géométriques simples, de crée d'un côté nos alliés avec une formes identiques mais également des bombes qui elles ont la même forme mais une taille variable.

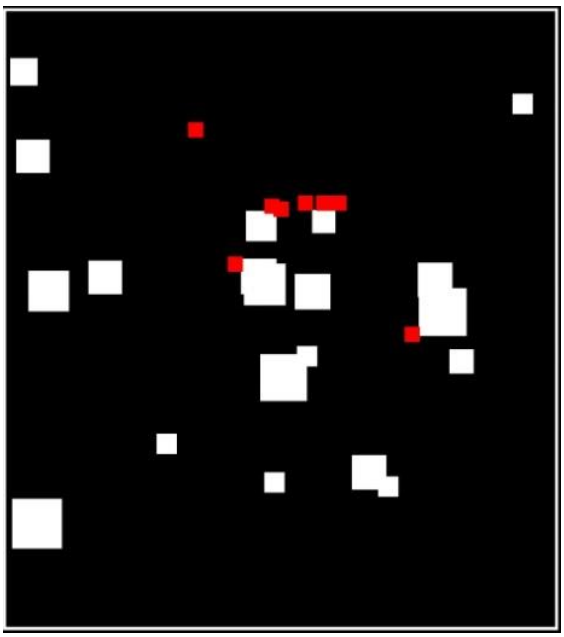

Figure 3: Jeu avant Textures

Un peu plus tard dans le projet, nous avons donc choisi le thème Pokémon pour notre jeu, nous avons donc utilisé une méthode (BufferedImage) permettant d'intégrer des images donnant au passage une identité à notre jeu.

Figure 4: Jeu après Textures

2.3/ Hitbox des personnages et pièges

Après avoir intégré le visuel de nos personnages il a fallu forcément leur créer une zone d'interaction (hitbox) autour de ses personnages sinon ils se passeraient simplement au travers l'un de l'autre et il en va de même pour les bordures de la zone de jeu.

Le plus simple pour les mettre en place a été de constamment comparer la position des alliés et de la comparer à celle des bordures et des bombes. Si la position est la même, on considère une collision et on tue l'allié concerné (tolérance 0, annihilation totale).

Figure 5: Alliés morts contre les hitbox (bombes et bordures)

2.4/ Implémentation du fitness

L'objectif de notre jeu étant d'atteindre le bas du tableau, notre fitness correspond donc simplement à la distance entre le joueur et le bas de l'écran. C'est une méthode simple pour intégrer ce principe malgré qu'il ait un défaut :

Selon le placement des bombes, un pseudo cul-de-sac pourrait se créer et si l'allié le plus bas est celui dans cette fameuse impasse, ça pourrait fausser la mutation et conduire à un blocage général des personnages…

Ce défaut peut être résolu via la mutation mais les chances sont assez minces.

Figure 6: Exemple d'impasse

2.5/ Implémentation de la génétique et des générations

Lors du lancement du jeu, on génère notre première Génération qui va faire son parcours de manière complétement aléatoire (aucune mutation…) et une fois qu'ils sont tous morts, on récupère la fitness du meilleur afin de l'isoler puisqu'il ne subira pas de mutation.

On fait ensuite un classement en commençant par le meilleur et on en fait une nouvelle liste. De cette liste on fait des couple (ou non c'est aléatoire) via une roue biaisée afin de faire des croisements entre les meilleurs éléments de la génération actuelle.

On fait ensuite une mutation et on en tire une nouvelle génération prête à parcourir le jeu et ce, jusqu'à obtenir un individu qui attendra le bas de l'écran au bout de « n » générations.

3/ Code

3.1/ Création de l'environnement

Pour commencer il a fallu créer la fenêtre graphique permettant l'affichage de notre jeu, dans lequel on a diviser l'écran en 2 panels via des Jpanel.

A noter que toutes les classes sont visible en Annexe.

```java
package com.perso.genetique.graphic;

import java.awt.BasicStroke;

public class PanelArene extends JPanel {

    public void paintComponent(Graphics g) {
        Graphics2D g2d = (Graphics2D) g;
        // Réinitialisation du background
        g.setColor(Color.BLACK);
        g.fillRect(0, 0, Frame.SCREEN_WIDTH/2, Frame.SCREEN_HEIGHT);

        // Aire du jeu
        g.setColor(Color.WHITE);

        g2d.setStroke(new BasicStroke(3));
        g.drawLine(Parameters.getLeftBorder(), Parameters.getTopBorder(), Parameters.getRightBorder(), Parameters.getTopBorder());
        g.drawLine(Parameters.getLeftBorder(), Parameters.getArrivee(), Parameters.getRightBorder(), Parameters.getArrivee());

        g.drawLine(Parameters.getLeftBorder(), Parameters.getTopBorder(), Parameters.getLeftBorder(), Parameters.getArrivee());
        g.drawLine(Parameters.getRightBorder(), Parameters.getTopBorder(), Parameters.getRightBorder(), Parameters.getArrivee());
    }
}
```

Figure 7: Jpanels

On a ensuite configuré cette fenêtre comme nous le souhaitions, à savoir par exemple :

- La taille de la fenêtre
- Nos panneaux gauches et droites
- Le titre et la couleur

On peut également voir qu'il y a également la fonction de lancement du jeu qui attend que l'utilisateur appuie sur le bouton « START ».

```java
package com.perso.genetique.graphic;

import java.awt.Color;

public class Frame extends JFrame implements ActionListener{

    public static int SCREEN_WIDTH = 1366;
    public static int SCREEN_HEIGHT = 728;

    private PanelLeft panLeft = null;
    private PanelArene panArene = new PanelArene();
    private PanelRight panRight = new PanelRight();

    public void frame() {

        this.setTitle("DeathRun Génétique");
        this.setSize(SCREEN_WIDTH, SCREEN_HEIGHT);

        this.setBackground(Color.BLACK);
        this.setLocation(200, 200);
        //this.setResizable(false);
        this.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        this.setLayout(new GridLayout(1, 2));

        PanelRight.startButton.addActionListener(this);

        this.getContentPane().add(panArene);
        this.getContentPane().add(panRight);
        this.setVisible(true);

        while(Parameters.START == false) {
            System.out.println("START : false");
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
            }
        }
```

Figure 8: Frame 1

On a ensuite créé la classe « Parameters » dans laquelle on stocke toutes nos variables nécessaires au jeu comme le nombre d'alliés, de bombes ou encore le taux de mutation. On y trouve également les conditions permettant à l'utilisateur de définir lui-même le nombre d'individus, de bombes et de génération.

```java
package com.perso.genetique.launcher;

import java.awt.Color;
import java.util.Random;

import com.perso.genetique.graphic.Frame;
import com.perso.genetique.graphic.PanelRight;

public class Parameters {
    private static int nGen = 0;
    private static int MAX_GEN;
    private static int NB_ALLIES;
    private static int NB_ALLIES_ALIVE = NB_ALLIES;

    private static int LIGNE_ARRIVEE = Frame.SCREEN_HEIGHT - 100;
    private static int LEFT_BORDER = 60;
    private static int RIGHT_BORDER = (Frame.SCREEN_WIDTH/2) - 80;
    private static int TOP_BORDER = 20;
    private static int SPAWN_X = (RIGHT_BORDER + LEFT_BORDER)/2;
    private static int SPAWN_Y = 140;
    private static int NB_BOMB;

    private static int bestfitness = 0;
    public static double crossoverRate;
    public static double mutationDeleteRate;
    public static double mutationAddRate;
    public static double mutationsRate;

    public static Random RANDOM = new Random();
    public static boolean FINISH = false;
    public static boolean START = false;
    public static boolean READ_NB_ALLIES = false;
    public static boolean POPULATION_CREATED = false;
```

Figure 9: Parameters 1

```java
        case 7:
            return Color.PINK;
        case 8:
            return Color.RED;
        case 9:
            return Color.WHITE;
        case 10:
            return Color.YELLOW;
        default:
            return Color.BLACK;
        }

    }

    public static void init() {

        if(!PanelRight.nA.getText().isEmpty())
            NB_ALLIES = Integer.parseInt(PanelRight.nA.getText());
        else
            NB_ALLIES = 10;

        if(!PanelRight.nB.getText().isEmpty())
            NB_BOMB = Integer.parseInt(PanelRight.nB.getText());
        else
            NB_BOMB = 20;

        if(!PanelRight.nG.getText().isEmpty())
            MAX_GEN = Integer.parseInt(PanelRight.nG.getText());
        else
            MAX_GEN = 100;
        }
}
```

Figure 10: Parameters 2

Bien plus tard dans le projet nous avons trouvé une identité à notre jeu et avons choisi le thème du jeu Pokémon. Il a donc fallu transformer nos jolis petits cubes et notre décor pour correspondre à ce thème.

Pour cela, on a utilisé la méthode « BufferedImage ». Pour les individus on a utilisé un case permettant, via un random, de sélectionner parmi une liste défini de texture d'individu correspondant à des Pokémon. Concernant les bombes, la même méthode a été utilisé mais sans case puisqu'il n'y a qu'une texture. Par contre, contrairement aux individus les bombes ont une taille qui varie.

```
private BufferedImage randomSkin() {

    try {
        Random r = new Random();
        int rnd = r.nextInt(16);
        switch(rnd) {
        case 0:
            skin = ImageIO.read(new File("img\\aquali.gif"));
            return skin;
        case 1:
            skin = ImageIO.read(new File("img\\arcko.gif"));
            return skin;
        case 2:
            skin = ImageIO.read(new File("img\\feunard.gif"));
            return skin;
        case 3:
            skin = ImageIO.read(new File("img\\gobou.gif"));
            return skin;
        case 4:
            skin = ImageIO.read(new File("img\\jirachi.png"));
            return skin;
        case 5:
            skin = ImageIO.read(new File("img\\killer-metamorph.png"));
            return skin;
        case 6:
            skin = ImageIO.read(new File("img\\miaouss.gif"));
            return skin;
        case 7:
            skin = ImageIO.read(new File("img\\abra.png"));
            return skin;
        case 8:
            skin = ImageIO.read(new File("img\\carapuce.png"));
            return skin;
        case 9:
            skin = ImageIO.read(new File("img\\hericendre.png"));
            return skin;
        case 10:
            skin = ImageIO.read(new File("img\\aspicot.png"));
            return skin;
```

Figure 11: liste des textures

On a du ensuite faire le panel de droite avec le menu configurable du jeu, pour cela on a utilisé d'autre fonctions de la methode « Jpanel » qui sont le bouton et la zone de texte (respectivement « Jbutton » et « JFormattedTextField ») nous permettant de récolter les entrées de l'utilisateur sur la configuration souhaitée.

```java
package com.perso.genetique.graphic;

import java.awt.BasicStroke;

public class PanelRight extends JPanel {

    public static JButton startButton = new JButton("START");
    public static JFormattedTextField nA = new JFormattedTextField(NumberFormat.getIntegerInstance());
    public static JFormattedTextField nG = new JFormattedTextField(NumberFormat.getIntegerInstance());
    public static JFormattedTextField nB = new JFormattedTextField(NumberFormat.getIntegerInstance());

    public void paintComponent(Graphics g) {

        this.setLayout(null);
        //this.setBackground(Color.BLACK);
        // Réinitialisation du background
        g.setColor(Color.BLACK);
        g.fillRect(0, 0, 960, 1040);

        Font police = new Font("Arial", Font.BOLD, 18);

        JLabel labA = new JLabel("Nombre d'alliés");
        labA.setFont(police);
        labA.setForeground(Color.WHITE);
        labA.setBounds(50, 320, 200, 20);
        nA.setBounds(50, 340, 150, 20);
        JLabel labG = new JLabel("Nombre de générations");
        labG.setFont(police);
        labG.setForeground(Color.WHITE);
        labG.setBounds(50, 420, 250, 20);
        nG.setBounds(50, 440, 150, 20);
        JLabel labB = new JLabel("Nombre de Bombes");
        labB.setFont(police);
        labB.setForeground(Color.WHITE);
        labB.setBounds(350, 320, 200, 20);
        nB.setBounds(350, 340, 150, 20);
        Dimension dStart = startButton.getPreferredSize();
        startButton.setBounds(50, 540, dStart.width + 100, dStart.height + 50);
```

Figure 12: Création panel droit

3.2/ Création des personnages et pièges

Une fois notre interface créée, il faut lui donner vie en créant nos individus et nos pièges.

Pour faire cela, nous avons créé une classe « Allies » héritant de la classe « Individual » et définie par plusieurs caractéristiques qui seront propres à chaque individu comme :

- Des coordonnées
- Une texture
- Un fitness
- Une liste de déplacement

Ensuite il nous a suffi d'intégrer les Alliés dans la méthode « PaintComponent » afin de les afficher à l'écran.

```java
public class Allies extends Individual {

    private List<Move> deplacement = new ArrayList<Move>();
    private int MOVE_INDEX = 0;
    private int fitness = 1000;
    private boolean HERITAGE = false;
    private boolean BEST = false;
    private BufferedImage skin = this.randomSkin();
    private Color color = Color.red;

    public void setBest(boolean b) {
        BEST = b;
    }

    public BufferedImage getSkin() {
        return skin;
    }

    private BufferedImage randomSkin() {
```

Figure 13: Allies 1

```java
public class Individual {

    protected boolean ALIVE = true;
    protected int ALLIES_SIZE = 20;
    protected int BOMB_SIZE;
    protected int POS_X = Parameters.getSpawnX();
    protected int POS_Y = Parameters.getSpawnY();
    //protected Color color = Parameters.randomColor();
    //protected Color color = Color.GREEN;

    public static Allies MVP = null;


    public Allies newAllies(Allies father, Allies mother) {

        List<Move> movement = new ArrayList<Move>();
        int r = Parameters.RANDOM.nextInt(100);

        if(r<50) { // Father premier
            int cut = Parameters.RANDOM.nextInt(father.getDepl().size());
            for(int i=0;i<=cut;i++) {
                movement.add(father.getDepl().get(i));
            }
            for(int i=cut;i<mother.getDepl().size();i++) {
                movement.add(mother.getDepl().get(i));
            }
        }
        else { // Mother première
            int cut = Parameters.RANDOM.nextInt(mother.getDepl().size());
            for(int i=0;i<=cut;i++) {
                movement.add(mother.getDepl().get(i));
            }
            for(int i=cut;i<father.getDepl().size();i++) {
                movement.add(father.getDepl().get(i));
            }
        }

        Allies all = new Allies(movement);
```

Figure 14: Individual 1

Pour les bombes le principe est le même, on crée la classe « Bombs » qui hérite de la classe « Individual » et ensuite on l'affiche à l'écran.

```java
public class Bomb extends Individual {

    private static int colorCount = 0;
    private static Color color =Color.WHITE;


    public int getColorCount() {
        return colorCount;
    }

    public void setColorCountIncr(int i) {
        colorCount+=i;
    }

    public void resetColorCount() {
        colorCount = 0;
    }

    public void setColor(Color c) {
        color = c;
    }

    public Color getColor() {
        return color;
    }

    public int getBombSize() {
        return BOMB_SIZE;
    }

    public void setBombSize(int i) {
        BOMB_SIZE = i;
    }
```

Figure 15: Bomb 1

3.3/ Hitbox des personnages et pièges

Certes, cette partie-là peut paraître complexe mais nous avons simplement fait en sorte de comparer la position des Individus à celle des bombes ainsi que des rebords puisqu'ils tuent également.

Donc si la position d'un individu est la même qu'une bombe ou un rebord, on considère l'individu comme étant mort.

```
        // On vérifie que l'allié ne touche pas les bords du cadre
        if((a.getPosX()<60 || a.getPosX()>860 - 15 || a.getPosY()<20) && a.getState() == true) {
            Parameters.setAlliesAlive(Parameters.getAlliesAlive() - 1);
            a.setState(false);
            continue;
        } // On vérifie que l'allié est vivant
        else if(a.getState() == false) {
            continue;
        } // On vérifie si un allié est arrivé
        else if(a.getPosY() + 15 > Parameters.getArrivee()) {
            Parameters.FINISH = true;
            Individual.MVP = a;
            panLeft.repaint();
            panRight.repaint();
            break aa;
        }
    }
```

Figure 16: Mort par rebord

```
    // On vérifie que l'allié ne touche pas les bombes
    for(Bomb b : EvolutionnaryProcess.getBomb()) {
        Rectangle rectA = new Rectangle(a.getPosX(), a.getPosY(), a.getAlliesSize(), a.getAlliesSize());
        Rectangle rectB = new Rectangle(b.getPosX(), b.getPosY(), b.getBombSize(), b.getBombSize());
        if(rectA.intersects(rectB) && a.getState() != false) {
            Parameters.setAlliesAlive(Parameters.getAlliesAlive() - 1);
            a.setState(false);

            // Animation en rouge pour voir quelle bombe s'est activée
            //b.setColor(Color.RED);
            continue;
        }
    }
```

Figure 17: Mort par bombe

3.4/ Implémentation de la génétique et des générations

Cette partie consiste en la gestion des déplacements et des générations des Alliés. La 1ère chose à faire est d'implémenter les déplacements de nos Individus et pour cela on a décidé de leur faire faire des déplacements aléatoires mais avec une légère augmentation de la probabilité de descendre. Cette modification était nécessaire pour éviter que les individus prennent une éternité à arriver à la fin.

Donc à la 1ère génération, ils se déplacent de manière aléatoire jusqu'à la fin de la génération et après la liste complète de leur déplacement est mise à jour avec leur parcours.

```
int x = a.getPosX();
int y = a.getPosY();
int depl= 0;
String dir;

// Déplacement aléatoire
int r = Parameters.RANDOM.nextInt(1000);
if(r<240) {
    depl = 3;
    //depl = Parameters.RANDOM.nextInt(4);
    y -= depl; // Déplacement vers le haut
    dir = "T";
}
else if(r<490) {
    depl = 3;
    //depl = Parameters.RANDOM.nextInt(4);
    x -= depl; // Déplacement vers la gauche
    dir = "L";
}
else if(r<740) {
    depl = 3;
    //depl = Parameters.RANDOM.nextInt(4);
    x += depl; // Déplacement vers la droite
    dir = "R";
}
else {
    depl = 3;
    //depl = Parameters.RANDOM.nextInt(4);
    y += depl; // Déplacement vers le bas
    dir = "B";
}

a.addDeplacement(dir, depl);

// Mise à jour des nouvelles positions
a.setPosX(x);
a.setPosY(y);
```

Figure 18: Frame mouvement

Lors de la 2nde génération, on récupère cette liste de déplacements et on y ajoute de nouveaux déplacements aléatoires qui correspondront aux déplacements de la génération actuelle. A noter qu'il faut également retirer les derniers mouvements de chaque individu sinon ils retourneraient mourir au même endroit !

```java
public class Move {

    private String direction;
    private int depl;

    public String setRandDir() {
        String dir;

        int r = Parameters.RANDOM.nextInt(1000);
        if(r<240)
            dir = "T";
        else if(r<490)
            dir = "L";
        else if(r<740)
            dir = "R";
        else
            dir = "B";

        return dir;
    }

    public String getDirection() {
        return direction;
    }

    public void setDirection(String s) {
        direction = s;
    }

    public int getDeplacement() {
        return depl;
    }

    public void setDeplacement(int i) {
        depl = i;
    }

}
```

Figure 19: Move

La dernière chose à faire pour cette partie consiste à gérer les générations, c'est-à-dire surveiller la génération actuelle et la remplacer lorsqu'ils sont tous morts. Pour cela il suffit d'une condition pour vérifier si la génération actuelle a fini de se déplacer afin de pouvoir faire les croisements puis recommencer.

```
aa:
    // Tant que pas arrivé en bas ou encore des générations à reproduire on continue
    while(Parameters.FINISH == false || Parameters.getGen() <= Parameters.getMaxGen()) {

        // Initialisation ou réinitialisation
        Parameters.setAlliesAlive(Parameters.getNbAllies());
        if(Parameters.getGen()>0) {
            Allies.spawnAllies(EvolutionnaryProcess.getPopulation());
            for(Allies a : EvolutionnaryProcess.getPopulation()) {
                a.resetMoveIndex();
                if(a.getDepl().size() > 50) {
                    for(int i=0;i<50;i++) {
                        a.getDepl().remove(a.getDepl().size()-1);
                    }
                }
            }
        }
```

Figure 20: Générations

3.5/ Implémentation du fitness et de la sélection génétique

Cette partie est la partie la plus compliquée, elle contient la gestion de fin de génération, à savoir :

- Calcul du fitness des Individus (distance individus→fin)
- La récupération du meilleur Individu
- Création d'une nouvelle liste en s'inspirant des meilleurs
- Croisement
- Mutation

On commence donc par évaluer la génération qui vient de se finir en comparant la position du bas de l'écran à leur position Y au moment de leur mort et on les classe du meilleur au moins bon puis on récupère le meilleur afin de l'isoler puisqu'il ne subira pas de croisement.

```
// On évalue toute la génération
for(int i=0;i<EvolutionnaryProcess.getPopulation().size();i++) {
    Allies a = EvolutionnaryProcess.getPopulation().get(i);
    a.evaluate();
    //System.out.println("Individu " + i + " : " + a.getFitness() + ", liste de dépl
}

// On récupère le meilleur de la population
EvolutionnaryProcess.getPopulation().sort(Comparator.comparing(Allies::getFitness));
//System.out.println("Le meilleur de cette génération a une fitness de " + Evolution

Individual.MVP = EvolutionnaryProcess.getPopulation(0);

Allies bestA = new Allies();
bestA.cloneDepl(EvolutionnaryProcess.getPopulation(0).getDepl());
bestA.setPosX(EvolutionnaryProcess.getPopulation(0).getPosX());
bestA.setPosY(EvolutionnaryProcess.getPopulation(0).getPosY());
bestA.setFitness();
//System.out.println("BEST : " + bestA.getFitness() + ", la taille de sa liste de mo
//bestA.setColor(Color.RED);
bestA.setHeritage(true);
bestA.setBest(true);

List<Allies> la = new ArrayList<Allies>();
la.add(bestA);
```

Figure 21: Classement fitness

Ensuite il faut faire le croisement de notre population père ou mère défini de manière aléatoire, on peut ensuite transmettre cela à la nouvelle génération.

```
for(int i=0;i<Parameters.getNbAllies()-1;i++) {
    // 1 ou 2 parents ?
    if(Parameters.RANDOM.nextDouble()<Parameters.crossoverRate) {

        Allies father = EvolutionnaryProcess.selection();
        Allies mother = EvolutionnaryProcess.selection();
        Allies newAllie = new Allies().newAllies(father, mother);
        //newAllie.setColor(Color.GREEN);

        la.add(newAllie);
    }
    else {
        Allies father = EvolutionnaryProcess.selection();
        Allies mother = EvolutionnaryProcess.selection();
        Allies newAllie = new Allies().newAllies(father, mother);
        //newAllie.setColor(Color.GREEN);

        la.add(newAllie);
    }
}



for(Allies al : la) {
    al.setHeritage(true);
    /*if(al.getBest() == true)
        al.setColor(Color.RED);
    else
        al.setColor(Color.GREEN);*/
}

// On transmet les caractéristiques à la nouvelles génération
EvolutionnaryProcess.survival(la);
Parameters.setGen(Parameters.getGen() + 1);
```

Figure 22: Croisement population

Enfin, la dernière chose à faire est la mutation, pour cela on génère un nombre random que l'on compare à notre taux de mutation pour savoir si l'on

```
public void mutate() {

    //MAJ
    //Suppression ?
    if(Parameters.RANDOM.nextDouble()<Parameters.mutationDeleteRate) {
        int moveIndex = Math.round(Parameters.RANDOM.nextInt(deplacement.size()*10)/10);
        for (int i =0; i < Math.round(deplacement.size()/5); i++) {
            if(moveIndex < deplacement.size())
                deplacement.remove(moveIndex);
        }
    }
    //Ajout ?
    if(Parameters.RANDOM.nextDouble()<Parameters.mutationAddRate) {
        int addIndex = Math.round(Parameters.RANDOM.nextInt(deplacement.size()*10)/10);
        List<Move> tmpArray = new ArrayList<Move>();
        for (int cpy = addIndex; cpy < deplacement.size(); cpy ++) {
            tmpArray.add(deplacement.get(cpy));
        }
        for (int i = 0; i < Math.round(deplacement.size()/5); i++) {
            Move newMove = new Move();
            newMove.setDirection(newMove.setRandDir());
            newMove.setDeplacement(Parameters.RANDOM.nextInt(4));
            if(addIndex < deplacement.size())
                deplacement.set(addIndex, newMove);
            else
                deplacement.add(newMove);
            addIndex++;
        }
        for (int j = 0; j < tmpArray.size(); j++) {
            if(addIndex < deplacement.size()) {
                deplacement.set(addIndex, tmpArray.get(j));
            }
            else {
                deplacement.add(tmpArray.get(j));
            }
            addIndex++;
        }
    }
    //Remplacement d'une section ?
    if(Parameters.RANDOM.nextDouble()<Parameters.mutationsRate) {
        int putIndex = Math.round(Parameters.RANDOM.nextInt(deplacement.size()*10)/10);
        for (int i = 0; i < Math.round(deplacement.size()/5); i++) {
            Move newMove = new Move();
            newMove.setDirection(newMove.setRandDir());
            newMove.setDeplacement(Parameters.RANDOM.nextInt(4));
            if(putIndex + Math.round(deplacement.size()/5) > deplacement.size()) {
                deplacement.add(newMove);
            }
            else {
                deplacement.set(putIndex, newMove);
                putIndex++;
            }
        }
    }
}
```

Figure 23: Mutation individus

4/ Conclusion

Ce projet a été une très bonne expérience pour nous car l'Algorithme Génétique est un moyen très concret et visualisable pour nous d'appréhender l'Intelligence Artificielle.

Le sujet que nous avons choisi pouvait sembler assez simple mais au final nous avons dû faire pas mal de modifications du support génétique fourni afin qu'il fonctionne au mieux avec notre jeu. Le fait de devoir modifier le support existant a rajouté une difficulté mais surtout un défi pour nos programmeurs objets. Nous avons d'ailleurs plusieurs idées à mettre en place afin de continuer à améliorer notre jeu.

Nous souhaiterions par exemple créer différentes cartes de jeu faites à l'avance (style labyrinthe par exemple…), également un système de téléportation auquel nous avions réfléchi pour créer une carte dans le genre « Escape room ». (voir annexe.)

La dernière chose mais pourtant la plus importante, nous pensons modifier la roue biaisée afin d'augmenter les chances de changement de gênes et permettre une vitesse de résolution accrue.

5/ Annexes

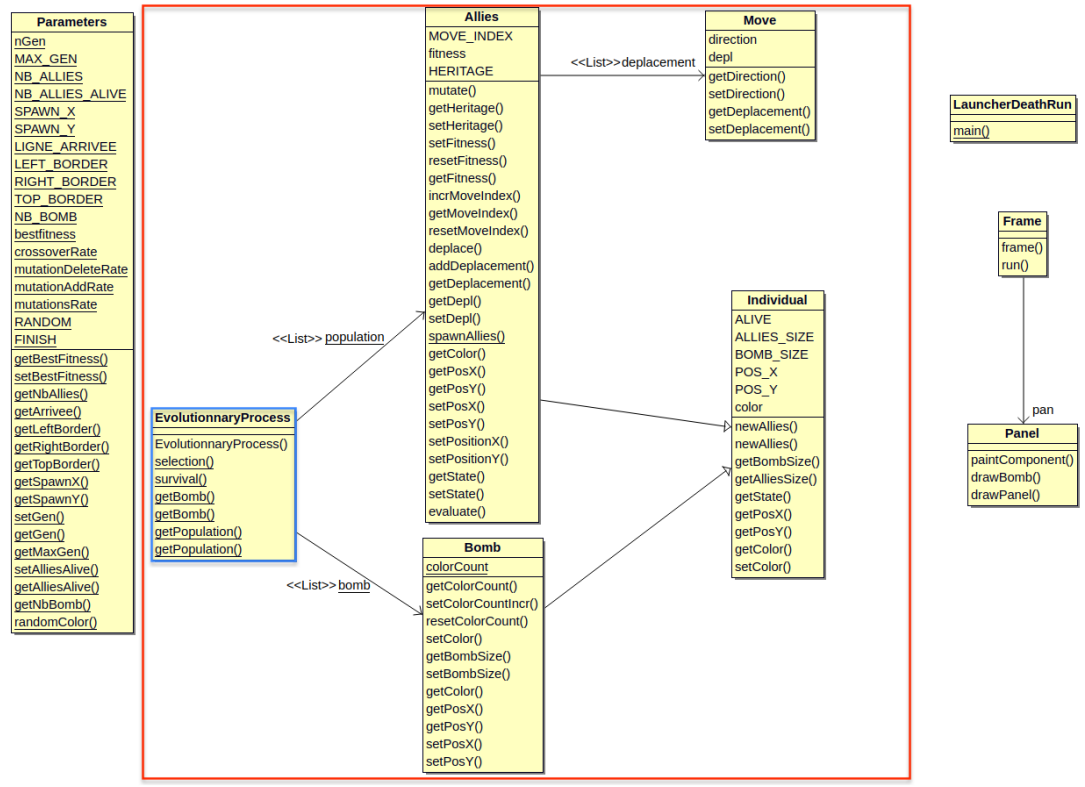

Figure 24: Organigramme

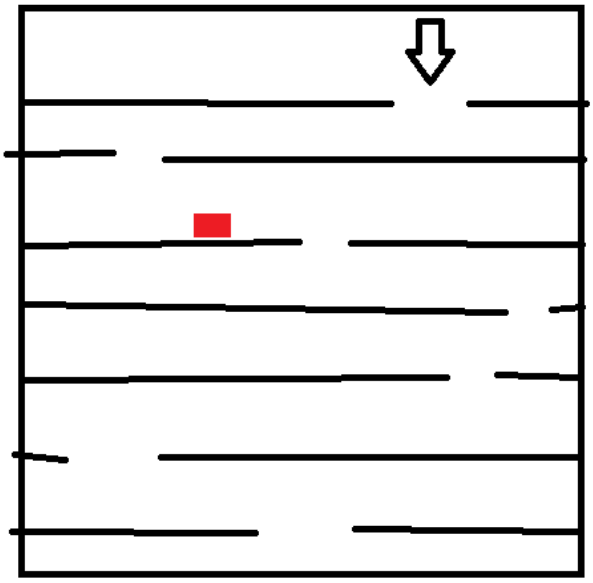

Figure 25: Map Donkey-Kong

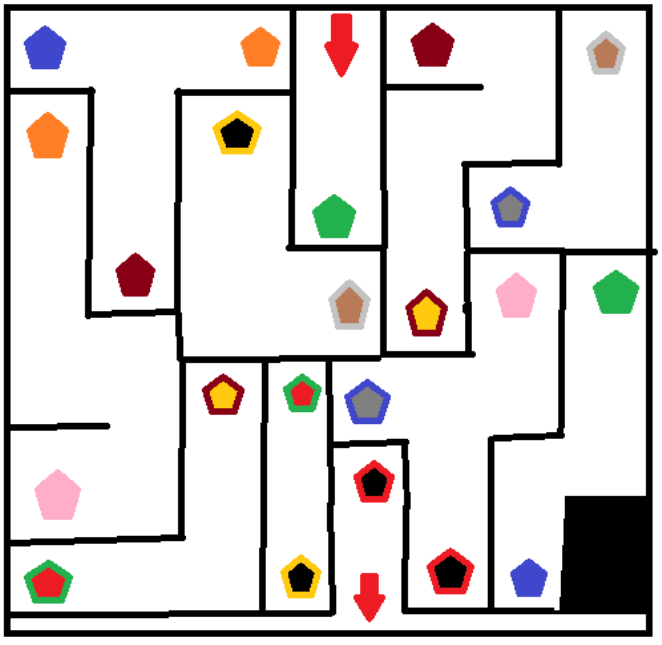

Figure 26: Map teleport