TP 5ème Année I.S.I : SMA

Banc de poissons 2D

Source : Livre L'intelligence Artificielle pour les développeurs

La première application est simulation d'un banc de poissons, inspirée des boids de Reynolds (https://fr.wikipedia.org/wiki/Boids) en 2D.

Nous allons voir un ensemble de poissons, représentés sous la forme de traits, se déplacer dans un océan virtuel et éviter des zones dangereuses à l'intérieur de celui-ci (cela peut représenter des obstacles physiques ou des zones présentant des prédateurs).

Le comportement du banc sera donc uniquement obtenu par émergence.

1. Les objets du monde et les zones à éviter

Avant de coder les agents eux-mêmes, nous allons coder une première classe qui peut être utilisée à la fois pour les objets et les agents. Celle-ci, nommée **Objet**, présente deux attributs ***posX*** et ***posY*** indiquant les coordonnées de l'objet.

Fichier : Objet.java

```
// Objet dans le monde (obstacle ou poisson)
public class Objet {
	public double posX;
	public double posY;

	public Objet() {
	}

	public Objet(double _x, double _y) {
		posX = _x;
		posY = _y;
	}

	public double Distance(Objet o) {
		return Math.sqrt((o.posX - posX) * (o.posX - posX) + (o.posY - posY) * (o.posY
- posY));
	}

	public double DistanceCarre(Objet o) {
		return (o.posX - posX) * (o.posX - posX) + (o.posY - posY) * (o.posY - posY);
	}
}
```

Les méthodes **Distance** et **DistanceCarre** permet de calculer la distance entre un l'objet et un autre objet de l'environnement.

Les zones à éviter **ZoneAEviter** sont des objets situés, qui possèdent la propriété ***rayon*** indiquant leur porté et le temps restant de vie de la zone noté ***tempsRestant***.

Fichier : ZoneAEviter.java

```
// Une zone à éviter pour les poissons. Celle-ci disparait automatiquement au bout d'un
moment
public class ZoneAEviter extends Objet {
	protected double rayon;
	protected int tempsRestant = 500;

	public ZoneAEviter(double _x, double _y, double _rayon) {
		posX = _x;
```

```
                posY = _y;
                rayon = _rayon;
        }

        public double getRayon() {
                return rayon;
        }

        public void MiseAJour() {
                tempsRestant--;
        }

        public boolean estMort() {
                return tempsRestant <= 0;
        }
}
```

2. Les agents-poissons

La classe **Poisson** représente les agents-poissons :

- La distance parcourue à chaque itération ***PAS*** en unité arbitraire.
- La distance indiquant la zone d'évitement ***DISTANCE_MIN*** et ***DISTANCE_MIN_CARRE***.
- La distance indiquant la zone d'alignement ***DISTANCE_MAX*** et ***DISTANCE_MAX_CARRE***.
- La direction des poissons est représenté par le déplacement en x codé via ***vitesseX*** et le déplacement en y codé via ***vitesseY*** à chaque itération.
- Le constructeur prend la position du départ et l'angle pris par le poisson.
- La méthode **MiseAJourPosition** permet de calculer la nouvelle position du poisson.
- La méthode **DansAlignement** permet de savoir si un autre poisson est proche.
- Les poissons doivent éviter de rentrer dans les autres poissons et aussi dans les murs. Les murs ne sont pas localisés en un point donné, la méthode **DistanceAuMur** calcule la plus petite distance par rapport aux murs.
- La vitesse d'un poisson est constante dans le temps, la méthode **Normaliser** permet de normaliser le vecteur vitesse.

Le comportement du poisson est :

- ✓ S'il y a un mur dans la zone très proche, alors on l'évite (**règle 1**).
- ✓ S'il y a une zone à éviter dans la zone très proche, alors on l'évite (**règle 2**).
- ✓ S'il y a un poisson dans la zone très proche, on s'en éloigne (**règle 3**).
- ✓ S'il y a un poisson dans la zone proche, on s'aligne sur lui (**règle 4**).

Fichier : Poisson.java

```
// Un poisson, géré par un agent
public class Poisson extends Objet {
        // Constantes
        public static final double PAS = 3;
        public static final double DISTANCE_MIN = 5;
        public static final double DISTANCE_MIN_CARRE = 25;
        public static final double DISTANCE_MAX = 40;
        public static final double DISTANCE_MAX_CARRE = 1600;

        // Attributs
        protected double vitesseX;
        protected double vitesseY;

        // Méthodes
```

```
	public Poisson(double _x, double _y, double _dir) {
		posX = _x;
		posY = _y;
		vitesseX = Math.cos(_dir);
		vitesseY = Math.sin(_dir);
	}

	public double getVitesseX() {
		return vitesseX;
	}

	public double getVitesseY() {
		return vitesseY;
	}

	protected void MiseAJourPosition() {
		posX += PAS * vitesseX;
		posY += PAS * vitesseY;
	}

	protected boolean DansAlignement(Poisson p) {
		double distanceCarre = DistanceCarre(p);
		return (distanceCarre < DISTANCE_MAX_CARRE && distanceCarre >
DISTANCE_MIN_CARRE);
	}

	protected double DistanceAuMur(double murXMin, double murYMin, double murXMax,
double murYMax) {
		double min = Math.min(posX - murXMin, posY - murYMin);
		min = Math.min(min, murXMax - posX);
		min = Math.min(min, murYMax - posY);
		return min;
	}

	protected void Normaliser() {
		double longueur = Math.sqrt(vitesseX * vitesseX + vitesseY * vitesseY);
		vitesseX /= longueur;
		vitesseY /= longueur;
	}

	protected boolean EviterMurs(double murXMin, double murYMin, double murXMax, double
murYMax) {
		// On s'arrête aux murs
		if (posX < murXMin) {
			posX = murXMin;
		} else if (posY < murYMin) {
			posY = murYMin;
		} else if (posX > murXMax) {
			posX = murXMax;
		} else if (posY > murYMax) {
			posY = murYMax;
		}

		// Changer de direction
		double distance = DistanceAuMur(murXMin, murYMin, murXMax, murYMax);
```

```
            if (distance < DISTANCE_MIN) {
                  if (distance == (posX - murXMin)) {
                        vitesseX += 0.3;
                  } else if (distance == (posY - murYMin)) {
                        vitesseY += 0.3;
                  } else if (distance == (murXMax - posX)) {
                        vitesseX -= 0.3;
                  } else if (distance == (murYMax - posY)) {
                        vitesseY -= 0.3;
                  }
                  Normaliser();
                  return true;
            }
            return false;
      }

      protected boolean EviterObstacles(ArrayList<ZoneAEviter> obstacles) {
            if (!obstacles.isEmpty()) {
                  // Recherche de l'obstacle le plus proche
                  ZoneAEviter obstacleProche = obstacles.get(0);
                  double distanceCarre = DistanceCarre(obstacleProche);
                  for (ZoneAEviter o : obstacles) {
                        if (DistanceCarre(o) < distanceCarre) {
                              obstacleProche = o;
                              distanceCarre = DistanceCarre(o);
                        }
                  }

                  if (distanceCarre < (obstacleProche.rayon * obstacleProche.rayon)) {
                        // Si collision, calcul du vecteur diff
                        double distance = Math.sqrt(distanceCarre);
                        double diffX = (obstacleProche.posX - posX) / distance;
                        double diffY = (obstacleProche.posY - posY) / distance;
                        vitesseX = vitesseX - diffX / 2;
                        vitesseY = vitesseY - diffY / 2;
                        Normaliser();
                        return true;
                  }
            }
            return false;
      }

      protected boolean EviterPoissons(Poisson[] poissons) {
            // Recherche du poisson le plus proche
            Poisson p;
            if (!poissons[0].equals(this)) {
                  p = poissons[0];
            } else {
                  p = poissons[1];
            }
            double distanceCarre = DistanceCarre(p);
            for (Poisson poisson : poissons) {
                  if (DistanceCarre(poisson) < distanceCarre && !poisson.equals(this)) {
                        p = poisson;
                        distanceCarre = DistanceCarre(p);
```

```
                }
            }

            // Evitement
            if (distanceCarre < DISTANCE_MIN_CARRE) {
                double distance = Math.sqrt(distanceCarre);
                double diffX = (p.posX - posX) / distance;
                double diffY = (p.posY - posY) / distance;
                vitesseX = vitesseX - diffX / 4;
                vitesseY = vitesseY - diffY / 4;
                Normaliser();
                return true;
            }
            return false;
        }

        protected void CalculerDirectionMoyenne(Poisson[] poissons) {
            double vitesseXTotal = 0;
            double vitesseYTotal = 0;
            int nbTotal = 0;
            for (Poisson p : poissons) {
                if (DansAlignement(p)) {
                    vitesseXTotal += p.vitesseX;
                    vitesseYTotal += p.vitesseY;
                    nbTotal++;
                }
            }
            if (nbTotal >= 1) {
                vitesseX = (vitesseXTotal / nbTotal + vitesseX) / 2;
                vitesseY = (vitesseYTotal / nbTotal + vitesseY) / 2;
                Normaliser();
            }
        }

        protected void MiseAJour(Poisson[] poissons, ArrayList<ZoneAEviter> obstacles,
double largeur, double hauteur) {
            if (!EviterMurs(0, 0, largeur, hauteur)) {
                if (!EviterObstacles(obstacles)) {
                    if (!EviterPoissons(poissons)) {
                        CalculerDirectionMoyenne(poissons);
                    }
                }
            }
            MiseAJourPosition();
        }
}
```

3. L'océan

L'environnement du banc de poissons est un océan virtuel. Il est mis à jour à la demande, de manière asynchrone. L'océan est un objet observable, qui préviendra tous les observateurs à chaque fin de mise à jour.

La classe **Ocean** va hériter de **Observable**. L'océan comporte un tableau de poissons, une liste d'obstacles, un générateur aléatoire et deux attributs pour indiquer la taille.

Fichier : Ocean.java

```
// L'océan dans lequel nagent les poissons
public class Ocean extends Observable {
	// Attributs
	protected Poisson[] poissons;
	protected ArrayList<ZoneAEviter> obstacles;
	protected Random generateur;
	protected double largeur;
	protected double hauteur;

	// Méthodes
	public Ocean(int _nbPoissons, double _largeur, double _hauteur) {
		largeur = _largeur;
		hauteur = _hauteur;
		generateur = new Random();
		obstacles = new ArrayList();
		poissons = new Poisson[_nbPoissons];
		for (int i = 0; i < _nbPoissons; i++) {
			poissons[i] = new Poisson(generateur.nextDouble() * largeur,
generateur.nextDouble() * hauteur,
					generateur.nextDouble() * 2 * Math.PI);
		}
	}

	public void AjouterObstacle(double _posX, double _posY, double rayon) {
		obstacles.add(new ZoneAEviter(_posX, _posY, rayon));
	}

	protected void MiseAJourObstacles() {
		for (ZoneAEviter obstacle : obstacles) {
			obstacle.MiseAJour();
		}
		obstacles.removeIf(o -> o.estMort());
	}

	protected void MiseAJourPoissons() {
		for (Poisson p : poissons) {
			p.MiseAJour(poissons, obstacles, largeur, hauteur);
		}
	}

	public void MiseAJourOcean() {
		MiseAJourObstacles();
		MiseAJourPoissons();
		setChanged();
		notifyObservers();
	}
}
```

La méthode **AjouterObstacle** crée une nouvelle zone à éviter aux coordonnées indiquées, avec la portée demandée et l'ajoute à la liste actuelle.

La mise à jour des obstacles consiste à demander à chaque zone de s'actualiser puis de supprimer les zones qui ont atteint leur fin de vie via la méthode **MiseAJourObstacles**.

La méthode **MiseAJourPoissons** effectue la mise à jour des poissons.

La méthode principale **MiseAJourOcean** est appelée depuis l'interface. On met à jour les obstacles puis les poissons, on indique que des changements ont eu lieu et on prévient les abonnés pour indiquer que cette mise à jour est terminée.

4. L'application graphique

Le programme principal est une application Swing.

Fichier : OceanPanel.java

```
// L'affichage de notre simulation
public class OceanJPanel extends JPanel implements Observer, MouseListener {
	protected Ocean ocean;
	protected Timer timer;

	public OceanJPanel() {
		this.setBackground(new Color(150, 255, 255));
		this.addMouseListener(this);
	}

	public void Lancer() {
		ocean = new Ocean(250, this.getWidth(), getHeight());
		ocean.addObserver(this);
		TimerTask tache = new TimerTask() {
			@Override
			public void run() {
				ocean.MiseAJourOcean();
			}
		};
		timer = new Timer();
		timer.scheduleAtFixedRate(tache, 0, 15);
	}

	protected void DessinerPoisson(Poisson p, Graphics g) {
		g.drawLine((int) p.posX, (int) p.posY, (int) (p.posX - 10 * p.vitesseX), (int)
(p.posY - 10 * p.vitesseY));
	}

	protected void DessinerObstacle(ZoneAEviter o, Graphics g) {
		g.drawOval((int) (o.posX - o.rayon), (int) (o.posY - o.rayon), (int) o.rayon *
2, (int) o.rayon * 2);
	}

	@Override
	public void update(Observable o, Object arg) {
		this.repaint();
	}

	@Override
	public void paintComponent(Graphics g) {
		super.paintComponent(g);
		for (Poisson p : ocean.poissons) {
			DessinerPoisson(p, g);
		}
		for (ZoneAEviter o : ocean.obstacles) {
			DessinerObstacle(o, g);
```

```
                }
        }

        @Override
        public void mouseClicked(MouseEvent e) {
                ocean.AjouterObstacle(e.getX(), e.getY(), 10);
        }

        @Override
        public void mousePressed(MouseEvent e) {
        }

        @Override
        public void mouseReleased(MouseEvent e) {
        }

        @Override
        public void mouseEntered(MouseEvent e) {
        }

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

Fichier : Application.java

```
// Lancement de la fenêtre et de l'application
public class Application {
        public static void main(String[] args) {
                // Création de la fenêtre
                JFrame fenetre = new JFrame();
                fenetre.setTitle("Banc de poissons");
                fenetre.setSize(600, 400);
                fenetre.setLocationRelativeTo(null);
                fenetre.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                fenetre.setResizable(false);
                // Création du contenu
                OceanJPanel panel = new OceanJPanel();
                fenetre.setContentPane(panel);
                // Affichage
                fenetre.setVisible(true);
                panel.Lancer();
        }
}
```