Université de Limoges
Faculté des Sciences & Techniques

Programmation Java (Processing)

Rapport de Projet : PAC-MAN

Réalisé par :
Rayane TAIBI

Encadré par :
M. Maxime MARIA

Année Universitaire 2025 - 2026

Plan du Rapport

Chapitre 1

Introduction

1.1 Contexte du projet

Dans le cadre de l'unité d'enseignement « Programmation Java (Processing) » offert à la Faculté des Sciences et Techniques de Limoges , il nous a été demandé d'accomplir un projet personnel qui consiste en la réalisation d'une version fonctionnelle du célèbre jeu d'arcade **Pac-Man**.

Ce projet a pour but de mettre en pratique les concepts de la programmation orientée objet, la gestion d'événements graphiques et la manipulation de fichiers au travers de la librairie Processing.

1.2 Objectifs

L'objectif principal du jeu est de contrôler un personnage (Pac-Man) à l'intérieur d'un labyrinthe afin de manger toutes les « Pac-Gommes » présentes, tout en évitant quatre fantômes qui se déplacent de manière autonome. Le jeu intègre un système de score, des bonus (Super-Pac-Gommes) permettant temporairement de manger les fantômes, ainsi qu'une gestion des vies

D'un point de vue technique, le développement devait respecter plusieurs contraintes imposées :

- L'utilisation du langage Java couplé à l'environnement **Processing**
- Le respect d'une architecture logicielle fournie, imposant l'utilisation de classes spécifiques telles que `Board`, `Hero` ou encore `Game`.
- La gestion de la persistance des données, notamment pour le chargement de la description du labyrinthe depuis un fichier texte et la sauvegarde de la partie en cours.
- L'implémentation d'un menu complet permettant de mettre le jeu en pause, de sauvegarder ou de consulter les meilleurs scores.

1.3 Organisation du rapport

Ce rapport décrit les différentes étapes de la réalisation de ce projet. Je commencerai par présenter l'analyse du problème ainsi que l'architecture technique choisie. Je détaillerai ensuite les fonctionnalités mises en place, depuis le déplacement du héros jusqu'au comportement des fantômes. Enfin, j'évoquerai les principaux obstacles rencontrés, les solutions adoptées et conclurai par les pistes d'amélioration possibles.

Chapitre 2

Architecture du Programme

Cette section présente l'organisation technique du projet. L'architecture modulaire imposée a été respectée rigoureusement, tout en intégrant les classes nécessaires à la gestion avancée des fantômes et du système de scores persistants.

2.1 Organisation des fichiers

Le code source est réparti dans plusieurs fichiers `.pde` (Processing). Cette séparation respecte le principe de responsabilité unique (Single Responsibility Principle), ce qui facilite grandement la maintenance et le débogage.

- **Fichiers sources (.pde) :**
  - `pacman.pde` : Point d'entrée du programme. Il contient les fonctions système `setup()` et `draw()`. Il initialise la fenêtre, charge les images en mémoire et lance la boucle principale.
  - `game.pde` : Le chef d'orchestre. Il instancie et relie toutes les entités (Hero, Board, Ghosts). C'est lui qui gère les transitions entre l'écran d'accueil, le jeu et le Game Over.
  - `board.pde` : Gère la logique du terrain. Il est responsable de la lecture des fichiers de niveau et de l'affichage des murs et des gommes.
  - `hero.pde` : Gère le personnage contrôlé par le joueur. Il contient la logique de déplacement fluide et la gestion du clavier.
  - `ghost.pde` : Gère les ennemis. Ce fichier a été ajouté pour centraliser toute la logique d'intelligence artificielle (IA) et les différents comportements des fantômes.
  - `menu.pde` : Gère exclusivement l'affichage des interfaces utilisateur (UI), comme le menu de pause ou le tableau des scores.
  - `constants.pde` : Un fichier utilitaire qui regroupe toutes les valeurs constantes (taille des cases en pixels, couleurs, vitesses, scores) afin de faciliter l'équilibrage du jeu sans fouiller dans le code.
- **Dossiers de ressources :**
  - `data/` : Contient les assets graphiques (sprites) et les fichiers de persistance (`savegame.txt` pour la sauvegarde, `highscores.txt` pour les scores).
  - `levels/` : Contient la description textuelle des niveaux (ex : `level1.txt`), ce qui permet de créer de nouveaux labyrinthes sans modifier le code source.

2.2 Détail de l'implémentation des classes

Nous détaillons ici le fonctionnement interne des classes principales et les choix algorithmiques effectués.

2.2.1 Classe Game : Le moteur d'états

La classe `Game` agit comme une machine à états finis.

— **Gestion des états :** Via l'énumération `GameState`, le jeu sait s'il est en mode `MENU`, `PLAYING` ou `GAMEOVER`. La méthode `update()` agit différemment selon cet état (ex : elle fige les calculs de position si le jeu est en pause).

— **Détection des collisions :** La méthode `checkCollisions()` est appelée à chaque image. Elle compare la distance euclidienne entre le centre de Pac-Man et le centre de chaque fantôme. Si cette distance est inférieure à la somme de leurs rayons (environ 15 pixels), une collision est validée.

— **Persistance des données :** La méthode `saveGame()` implémente une sérialisation manuelle. Elle écrit l'état du jeu dans un fichier texte ligne par ligne : d'abord les métadonnées (score, vies), puis l'état complet de la grille (converti en chaîne de caractères), et enfin la position vectorielle de chaque entité.

2.2.2 Classe Board : La structure du niveau

Le plateau n'est pas une simple image, mais une grille logique interactive.

— **Parsing du niveau :** La méthode `chargeLevel()` lit le fichier texte caractère par caractère. Chaque symbole est interprété pour remplir le tableau 2D `_cells[][]` : un '' crée un mur, un '.' une gomme, etc.

— **Conversion Coordonnées :** La classe fournit des méthodes essentielles pour traduire une position en pixels (x, y sur l'écran) vers des coordonnées de grille ($colonne, ligne$). Cela permet aux entités de savoir "sur quelle case" elles se trouvent réellement.

2.2.3 Classe Hero : Déplacement et Anticipation

Le déplacement du joueur a fait l'objet d'une attention particulière pour assurer la fluidité.

— **Mouvement pixel par pixel :** Contrairement à une version simplifiée qui sauterait de case en case, Pac-Man possède une position précise (PVector). Sa vitesse est ajoutée à sa position à chaque frame.

— **Buffer d'entrée (Next Direction) :** Pour éviter la frustration du joueur, nous utilisons un attribut `_nextDirection`. Si le joueur appuie sur une touche avant d'arriver à une intersection, l'intention est mémorisée. Dès que le centre de Pac-Man s'aligne avec l'intersection et que le mur n'existe pas, le virage est exécuté automatiquement. Cela rend le contrôle beaucoup plus souple.

2.2.4 Classe Ghost : Intelligence Artificielle

L'IA des fantômes repose sur un système de cibles dynamiques.

— **Différenciation des comportements :** La méthode `updateTarget()` assigne une case cible différente selon le type de fantôme :
  — *Blinky* vise la position exacte de Pac-Man.
  — *Pinky* vise 4 cases devant Pac-Man (anticipation).
  — *Clyde* change de comportement selon sa distance avec le joueur.
  — *Inky* utilise une cible complexe dépendant de la position de Blinky.

— **Algorithme de Pathfinding :** À chaque intersection, le fantôme évalue ses voisins. Il exclut la case d'où il vient (interdiction de faire demi-tour) et calcule la distance à vol d'oiseau vers sa cible pour les cases restantes. Il choisit alors la direction qui minimise cette distance.

2.3 Diagramme de Classes UML

La figure ci-dessous présente l'architecture orientée objet du projet. **Note importante sur la représentation :** Par souci de lisibilité, ce diagramme est une version synthétique. Il ne contient que les attributs et méthodes essentiels à la compréhension de la logique globale du projet (les getters, setters et méthodes utilitaires mineures ont été volontairement omis). Il met en évidence les relations de composition (losanges pleins) et de dépendance (flèches en pointillés) entre les modules.

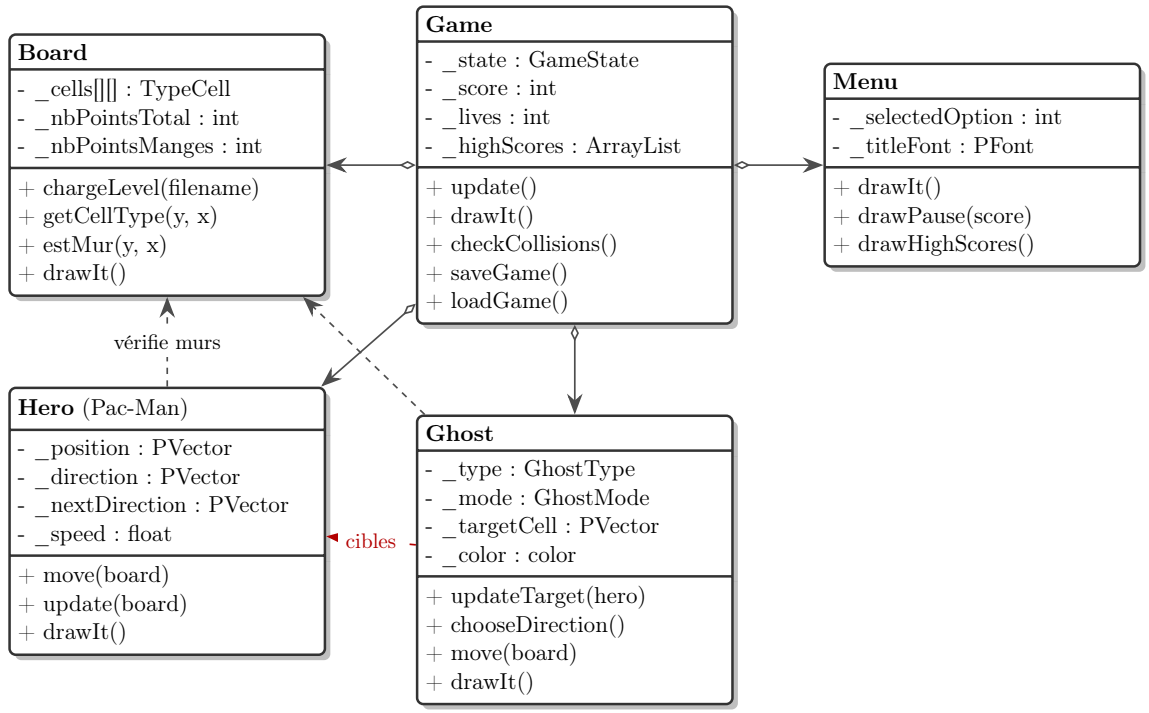

FIGURE 2.1 – Diagramme UML synthétique : seules les méthodes principales et les variables critiques pour la logique du jeu sont représentées.

Chapitre 3

Difficultés techniques et Solutions

Ce chapitre détaille les obstacles majeurs que j'ai rencontrés durant le développement, ainsi que les stratégies de contournement ou les correctifs appliqués.

3.1 Gestion du Rendu Graphique (Sprites vs Primitives)

Problème

Initialement, je souhaitais utiliser des « sprites » (images) pour Pac-Man et les fantômes. Cependant, l'intégration a causé des soucis de performance et de transparence.

Solution

J'ai opté pour un rendu vectoriel procédural. Je dessine les entités en temps réel avec les fonctions graphiques de Processing. Par exemple, pour Pac-Man, j'utilise la fonction `arc()` en faisant varier l'angle pour simuler la bouche qui s'ouvre et se ferme :

```
// hero.pde : Gestion de l'ouverture de la bouche
fill(255, 255, 0); // Jaune
noStroke();
// Dessine un camembert partiel selon l'angle de la bouche
arc(0, 0, _size, _size, _angleBouche, TWO_PI - _angleBouche, PIE);
```

Listing 3.1 – Dessin procédural de Pac-Man (hero.pde)

Pour les fantômes, j'ai combiné un rectangle et une ellipse, avec une couleur dynamique définie par l'état du fantôme :

```
// ghost.pde : La couleur change si le fant me est effray
if (_mode == GhostMode.FRIGHTENED) {
  fill(33, 33, 222); // Bleu fonc
} else {
  fill(_color); // Couleur normale du fant me
}
// Corps du fant me (Haut arrondi + Bas rectangulaire)
arc(0, 0, _size, _size, PI, TWO_PI, CHORD);
rect(-_size/2, 0, _size, _size/2);
```

Listing 3.2 – Dessin dynamique des fantômes (ghost.pde)

3.2 Logique de sortie de la « Maison des Fantômes »

Problème

Les fantômes démarrent dans une cage entourée de murs. Le moteur de collision standard les empêchait de sortir.

Solution

J'ai créé un mode spécifique `LEAVING_HOME`. Dans ce mode, je définis une cible située au-dessus de la cage et j'autorise temporairement le passage à travers les murs.

```
// ghost.pde : D finition de la cible vers la sortie
if (_mode == GhostMode.LEAVING_HOME) {
  // On vise une cellule au-dessus pour sortir
  _targetCell = new PVector(_cellX, _cellY - 5);

  // Une fois dehors (plus de VOID autour), on passe en CHASE
  if (board.getCellType(_cellY, _cellX) != TypeCell.VOID) {
     _mode = GhostMode.CHASE;
  }
}
```

Listing 3.3 – Logique de sortie de la cage (ghost.pde)

3.3 Retour à la base après capture

Problème

Lorsqu'un fantôme est mangé, il doit retourner rapidement à sa cage sans rester bloqué dans les couloirs du labyrinthe.

Solution

J'ai implémenté le mode `EATEN`. Dans cet état, le fantôme ignore les collisions, accélère (vitesse x 1.5) et file tout droit vers son point de spawn.

```
// ghost.pde : Acc l ration et cible vers le spawn
if (_mode == GhostMode.EATEN) {
  currentSpeed = _speed * 1.5; // Plus rapide pour rentrer
  _targetCell = new PVector(_spawnX, _spawnY); // Cible = Cage
}

// Autorisation de traverser les murs en mode EATEN
boolean canMove = (_mode == GhostMode.EATEN) || !board.isWallForGhost(
    nextY, nextX);
```

Listing 3.4 – Gestion du retour à la base (ghost.pde)

3.4 Passages d'un bord à l'autre (Tunneling)

Problème

Le jeu original Pac-Man permet au personnage de sortir d'un côté du labyrinthe pour réapparaître instantanément de l'autre (le « tunneling »). Lors de mes tentatives d'implémentation, cette mécanique provoquait systématiquement des erreurs de type `ArrayIndexOutOfBoundsException` dans la gestion du tableau 2D, car le personnage tentait d'accéder à des coordonnées situées en dehors des limites du tableau.

Solution

Malgré plusieurs essais pour gérer la téléportation du personnage d'un bord à l'autre, je n'ai pas réussi à trouver une solution stable qui ne provoque pas l'arrêt brutal du programme.

Par manque de temps et pour garantir que le jeu ne plante pas lors de la démonstration, j'ai pris la décision de ne pas inclure le tunneling dans la version finale. J'ai donc considéré les bords du labyrinthe comme des limites infranchissables.

3.5 Intelligence Artificielle du fantôme Blinky

Problème

Pour le comportement de chasse, j'ai programmé le fantôme Blinky pour qu'il se dirige toujours vers la case de Pac-Man. Cependant, j'ai constaté un bug récurrent : le fantôme se retrouve parfois coincé dans des configurations de murs en forme de « U ». Il essaie indéfiniment d'avancer vers Pac-Man à travers le mur car c'est le chemin le plus court géométriquement, sans comprendre qu'il doit contourner l'obstacle.

Solution

Je n'ai pas réussi à corriger ce défaut dans la version finale. J'ai identifié qu'il faudrait implémenter un algorithme de recherche de chemin plus complexe (comme A* ou Dijkstra) pour que le fantôme puisse anticiper les murs et trouver le vrai chemin le plus court.

Faute de temps pour développer cette solution avancée, le comportement est resté en l'état. Le fantôme arrive généralement à se décoincer lorsque Pac-Man se déplace suffisamment pour changer l'angle de poursuite.

FIGURE 3.1 – Illustration du blocage de l'IA (minimum local)

Conclusion

En conclusion, ce projet a permis d'atteindre l'objectif principal : développer une version fonctionnelle et jouable de *Pac-Man* en Java. Le jeu intègre les mécaniques essentielles (déplacements, collisions, gestion des vies) ainsi que des fonctionnalités techniques comme la persistance des scores et la sauvegarde de la partie.

Ce travail a été une excellente mise en pratique de la programmation orientée objet. J'ai pu comprendre l'importance d'une architecture claire pour gérer les interactions entre le plateau, le héros et les fantômes.

Cependant, certaines limites subsistent. Comme expliqué dans le rapport, l'intelligence artificielle du fantôme rouge rencontre parfois des blocages face aux murs et le passage d'un bord à l'autre (tunneling) a été retiré pour garantir la stabilité de la démo. Si le projet devait continuer, la priorité serait d'implémenter un algorithme de recherche de chemin plus complexe (comme A*) pour rendre les ennemis plus malins, et d'ajouter du son pour l'ambiance.

Globalement, ce projet m'a permis de progresser en algorithmie et de livrer un programme structuré, capable de gérer des données complexes.