HEIG-VD — MCR

Projet (Visitor) – Rapport d'implémentation

Loïc Herman, Samuel Roland, Massimo Stefani, Timothée Van Hove

16 juin 2024

1 Introduction

Ce projet consiste en la création d'un jeu inspiré de Pacman, où le joueur contrôle un personnage devant naviguer dans un labyrinthe, collecter des pellets et éviter les fantômes. Le jeu améliore le comportement d'un Pacman classique car les fantômes ont des comportements variés et des capacités spéciales. L'objectif principal du joueur est de collecter tous les pellets tout en évitant d'être mangé par les fantômes.

Le but de ce rapport est de fournir une explication détaillée de l'architecture du projet, des détails de son implémentation, et de la manière dont les différents composants interagissent pour former le jeu final.

2 Instructions de compilation

Notre projet utilise Java 21 et Gradle, le wrapper Gradle est inclus dans les sources du programme. La suite Swing est utilisée donc aucune autre dépendance est nécessaire.

2.1 Compilation du projet

Pour compiler le projet, la commande `./gradlew build` doit être exécutée à la racine des sources fournies. Cette opération peut prendre un peu de temps lors de la première exécution, car le daemon doit être initialisé.

2.2 Exécution du programme

L'exécution peut être faite soit en exécutant manuellement le fichier Jar, soit en lançant la commande `./gradlew run` qui démarrera automatiquement une instance du jeu.

3 Contexte de mise en œuvre

La conception de notre jeu s'inspire de la structure classique d'un jeu 2D, avec une séparation claire entre les différentes responsabilités. Les principaux composants de notre projet incluent :

- **GameWindow** : Gère la fenêtre principale du jeu et orchestre le cycle de vie du jeu.
- **Level** : Représente un niveau du jeu, contenant la logique de jeu principale, les entités et la gestion des interactions.
- **Player** : Représente le joueur contrôlé par l'utilisateur, avec des états et comportements variés.

- **Ghost** : Représente les ennemis du jeu, chacun avec des comportements distincts.
- **MapParser et LevelBuilder** : Responsables de la génération et de la construction des niveaux à partir des fichiers de cartes.
- **Sprite et ses sous-classes** : Gèrent l'affichage graphique des entités.
- **InteractionVisitor** : Gère les interactions entre les différentes entités du jeu.

Les interactions entre ces composants sont gérées principalement par le `GameWindow`, qui assure la coordination entre l'affichage (`GameFrame` et `LevelPanel`) et la logique de jeu (`Level`, `Player`, `Ghosts`).

3.1 Initialisation et Lancement du jeu

Les sprites sont chargées en mémoire à partir des fichiers de ressources. Cela se fait principalement dans les classes de gestion des sprites comme `PacmanSprites`. Le niveau du jeu est construit en utilisant `MapParser` et `LevelBuilder`. Ces classes lisent la carte du niveau à partir d'un fichier texte et créent les cellules, les entités et les configurations initiales. Les joueurs, les fantômes et les pellets sont créées et placées dans leurs positions initiales sur le tableau de jeu. La méthode `main` appelle `GameWindow.getInstance().begin()` pour démarrer le jeu.

La classe `GameWindow` utilise le patron `Observer` pour réagir aux événements du niveau. Elle implémente l'interface `Level.LevelObserver` et s'inscrit en tant qu'observateur du niveau. Lorsque le niveau signale une victoire ou une défaite, `GameWindow` reçoit la notification et arrête le jeu tout en affichant un message à l'utilisateur.

Chaque entité mobile (joueur ou fantôme), est lancée dans un thread séparé pour gérer ses mouvements de manière asynchrone. Un `ScheduledExecutorService` est dédié à chaque entité, ce qui permet de planifier l'exécution périodique des tâches de mouvement. Lors du démarrage du jeu avec la méthode `start()`, un service d'exécution est créé pour chaque entité et une tâche de mouvement (`EntityTask`) est planifiée pour s'exécuter à intervalles réguliers, déterminés par l'intervalle de déplacement de chaque entité. Les threads sont arrêtés lorsque le jeu est mis en pause ou terminé, en appelant la méthode `stop()` qui arrête tous les services d'exécution en cours.

3.2 Construction de la map

La map est structurée comme un tableau 2D de cellules, chaque cellule représentant une partie distincte du terrain du jeu. Ce tableau 2D permet de modéliser les différentes zones du jeu, comme que les murs, les chemins et les zones de collecte. Chaque cellule est une instance d'une sous-classe de `Cell`, ce qui permet de définir des comportements spécifiques pour différents types de terrains, comme les murs infranchissables.

La classe `MapParser` est responsable de lire les données de la carte à partir d'un fichier et de les interpréter pour construire le tableau 2D de cellules. Le fichier de carte contient des caractères représentant différents éléments du jeu.

Après que `MapParser` a interprété le fichier de carte `LevelBuilder` utilise ces informations pour instancier les cellules et les entités appropriées. Il crée des instances de cellules comme `WallCell`,

`GroundCell`, et `DoorCell`, et les place aux positions correspondantes dans le tableau 2D de cellules. `LevelBuilder` instancie aussi les entités telles que les joueurs et les fantômes, en les plaçant dans les cellules appropriées.

3.3 Rendering

La classe `LevelPanel` est responsable de dessiner le tableau de jeu et toutes les entités qui s'y trouvent. Lorsqu'elle est appelée, `paintComponent` rend chaque cellule du tableau de jeu, en utilisant la méthode `render` qui parcourt toutes les cellules et appelle `renderCell` pour dessiner le contenu de chaque cellule. `renderCell` utilise l'interface `Sprite` pour dessiner les entités présentes dans la cellule, en tenant compte de leur couche de rendu afin de s'assurer que les entités sont dessinées dans le bon ordre.

Plusieurs implémentations de L'interface `Sprite` existent pour gérer différents types de rendu. `ImageSprite` est utilisé pour dessiner des images statiques, tandis que `AnimatedSprite` gère les animations en affichant successivement une série de frames à un intervalle donné. Chaque sprite sait comment se dessiner sur le contexte graphique fourni, en tenant compte de sa position et de sa taille. Les classes de sprites sont responsables de découper les images en sous-images si nécessaire, et de gérer les détails de l'affichage tels que le clignotement ou les transitions d'animation.

Les entités qui nécessitent un rendu spécial implémentent l'interface `SpecialRender`, qui définit des méthodes pour vérifier si un rendu spécial est nécessaire et pour effectuer ce rendu. Par exemple, l'animation de téléportation du StormTrooper est gérée par cette interface. Lorsqu'un `StormTrooper` est en cours de téléportation, `renderSpecial` est appelé pour dessiner l'animation de téléportation au-dessus de l'entité. Cela permet d'intégrer de manière transparente des effets visuels complexes sans alourdir la logique de dessin des entités elles-mêmes.

3.4 Déplacement des NPC

L'algorithme de pathfinding utilisé pour les mouvements des fantômes est basé sur BFS. En explorant toutes les cellules voisines de manière systématique, l'algorithme construit un chemin optimal en termes de nombre de déplacements nécessaires pour atteindre la cible.

La logique de mouvement des fantômes est gérée par la méthode `getNextMove` qui détermine la direction que doit prendre un fantôme à chaque étape. Lorsque le fantôme doit se déplacer, la méthode `getNextMove` est appelée pour calculer sa prochaine direction basée sur son chemin actuel. Si le fantôme n'est pas en train de retourner à sa maison, la méthode met à jour le chemin vers le joueur à intervalles réguliers, définis par un compteur de mouvements. Si le fantôme est en mode "retour à la maison" (souvent après avoir été mangé par le joueur en mode invincible), il met à jour son chemin vers sa cellule de départ. Cette mise à jour du chemin se fait en utilisant l'algorithme de pathfinding pour recalculer le chemin le plus court.

En plus de la mise à jour du chemin, la méthode `getNextMove` intègre également une logique pour gérer les mouvements aléatoires lorsque le fantôme n'a pas de chemin spécifique à suivre.

3.5 Gestion des interactions

Le mécanisme d'interaction entre les entités dans le jeu repose sur le modèle Visitor.

L'interface `InteractionVisitor` définit une méthode de visite pour chaque type d'entité avec laquelle une interaction peut avoir lieu. Lorsqu'une entité doit interagir avec une autre, elle accepte un visiteur qui implémente l'interface `InteractionVisitor`. Ce visiteur contient la logique spécifique de ce qui doit se passer lors de l'interaction. Par exemple, lorsqu'un joueur rencontre un pellet, le visiteur approprié sera utilisé pour gérer la collecte du pellet.

Les interfaces `Interactor` et `Interactable` définissent les rôles respectifs des entités pouvant initier et recevoir des interactions. Une entité qui implémente l'interface `Interactor` est capable d'interagir avec d'autres entités. Elle contient la méthode `interactWith` qui est appelée pour déclencher une interaction. D'autre part, les entités qui implémentent l'interface `Interactable` peuvent être la cible d'interactions. Elles doivent implémenter la méthode `accept`, qui accepte un `InteractionVisitor`.

Ce design permet une grande flexibilité dans la définition des comportements des entités. Par exemple, un fantôme (qui est un `Interactor`) peut interagir avec un joueur (qui est un `Interactable`) en utilisant un `InteractionVisitor` pour définir ce qui se passe lorsque le fantôme rencontre le joueur. De cette manière, les interactions complexes peuvent être gérées proprement et efficacement, en permettant aux développeurs d'ajouter ou de modifier des comportements d'interaction sans avoir à modifier directement les classes des entités concernées.

3.5.1 États du joueur

Le pattern `State` est utilisé pour gérer les différents états du joueur dans le jeu. Ce qui permet de représenter chaque état possible du joueur dans des classes distinctes. `PlayerNormalVisitor` représente l'état normal du joueur où il est vulnérable aux attaques des fantômes. `PlayerInvincibleVisitor` représente l'état invincible du joueur, déclenché lorsqu'il mange un super pellet. Dans cet état, le joueur peut manger les fantômes, les envoyant ainsi à leur point de départ, et il devient invulnérable aux attaques des fantômes. `PlayerDeadVisitor` est l'état dans lequel le joueur se trouve lorsqu'il est tué par un fantôme. Dans cet état, le joueur ne peut plus interagir avec l'environnement et affiche l'animation de mort du Pacman.

Les transitions entre les états sont déclenchées par des événements spécifiques. Par exemple, lorsque le joueur mange un super pellet, l'état passe de `PlayerNormalVisitor` à `PlayerInvincibleVisitor`. Cette transition est gérée en changeant le gestionnaire d'état (`handler`) du joueur vers une instance de `PlayerInvincibleVisitor`. De même, si un fantôme attrape le joueur lorsqu'il est en état normal, le gestionnaire d'état est changé pour `PlayerDeadVisitor`.

4 Utilisation du pattern Visiteur dans le cadre des interactions

Le mécanisme d'interaction entre les entités dans le jeu repose sur le modèle Visitor. Ce modèle permet une grande flexibilité dans la définition des comportements des entités en séparant la

logique des interactions de la structure des objets. Les interactions dans notre jeu se décomposent en plusieurs scénarios en fonction des états des entités et des types de rencontre.

Nous avons cherché à implémenter différents types d'interactions car nous voulions diversifier non seulement le comportement lors d'une interaction, mais également le sens et les changements possibles dépendamment du gestionnaire (*handler*) utilisé.

4.1 Player en mode normal

Lorsque le joueur est en mode normal, deux interactions principales peuvent se produire :

- **Player → Super Pellet** : Lorsque le joueur rencontre un super pellet, il le mange et passe en mode invincible. Cela déclenche une transition d'état qui modifie le comportement du joueur.
- **Player → Boba Fett** : Si le joueur rencontre Boba Fett, cela applique un boost de vitesse à Boba Fett, modifiant ainsi temporairement son comportement.

4.2 Player en mode invincible

En mode invincible, le joueur a la capacité d'interagir avec un plus grand nombre d'entités, et les interactions sont plus variées :

- **Player → Super Pellet** : Manger un super pellet prolonge le timer de l'invincibilité du joueur.
- **Player → Boba Fett** : Le joueur prend la vitesse de Boba Fett et le renvoie à la case départ, ce qui réinitialise la position de Boba Fett dans le labyrinthe.
- **Player → Luke** : Lorsque le joueur essaie de manger Luke, cela se traduit par la mort du joueur, car Luke est une entité spéciale. Veuillez noter que la mort du Pacman lors de l'interaction avec Luke se produit car le Pacman déclenche l'interaction avec lui.
- **Player → Sith** : Rencontrer un Sith envoie ce dernier à la maison, neutralisant temporairement la menace qu'il représente.
- **Player → Storm Trooper** : Le joueur peut retirer un Storm Trooper du jeu en le rencontrant, éliminant ainsi cette menace.
- **Player → Vader** : Rencontrer Vader renvoie ce dernier à la maison et change la couleur du Pacman en rouge, signalant une modification visuelle de l'état du jeu.

4.3 Interactions des fantômes

Les fantômes ont également leurs propres interactions avec les autres entités, en fonction de leurs comportements individuels :

- **Boba Fett → Super Pellet** : Lorsque Boba Fett mange un super pellet, une pensée apparaît, illustrant une interaction unique pour ce personnage.
- **Sith → Storm Trooper** : Un Sith peut téléporter un Storm Trooper à une position aléatoire dans le labyrinthe, ce qui ajoute une dynamique imprévisible aux mouvements des ennemis.

- **Vader → Luke** : Quand Vader rencontre Luke, un dialogue s'installe entre les deux, ajoutant une couche narrative à leur interaction.
- **Fantômes → Player** : Si un fantôme rencontre le joueur en mode normal, le joueur est tué, ce qui entraîne la fin de la partie pour celui-ci.

5 Conclusion

Ce projet a été une expérience enrichissante qui nous a permis d'explorer et de mettre en œuvre des patterns de conception pour développer notre jeu inspiré de Pacman. Nous avons apprécié la simplicité apportée par le pattern Visiteur, qui a rendu la gestion des interactions entre les entités du jeu plus élégante et efficace. Grâce à ce pattern, nous avons pu définir des comportements spécifiques pour chaque interaction sans avoir à modifier les classes des entités elles-mêmes, ce qui a rendu notre code plus flexible et extensible.

L'utilisation du pattern State a été utilisée pour représenter chaque état du joueur par une classe distincte, ce qui nous a permis de simplifier la gestion des transitions d'état et assuré une meilleure organisation du code.

Nous avons également constaté les avantages du principe de l'OCP, qui a guidé notre conception. Cette approche a permis d'intégrer de nouvelles fonctionnalités, sans compromettre l'extensibilité du code.

En somme, ce projet a démontré l'importance des patterns de conception en pratique comme Visiteur et State, pour la création de logiciels extensibles et maintenables. Nous sommes fiers de notre projet et de l'architecture logicielle que nous avons conçue.