Projet de Programmation - Rapport

AHMED YAHIA Yacine
BOUZAIENNE Feriel
DEBBAH Martin
PENAS BORTOLUZZI Dorian
POUILLART Lucas

Pour le 13 Mai 2022

Contents

1 Introduction et présentation générale

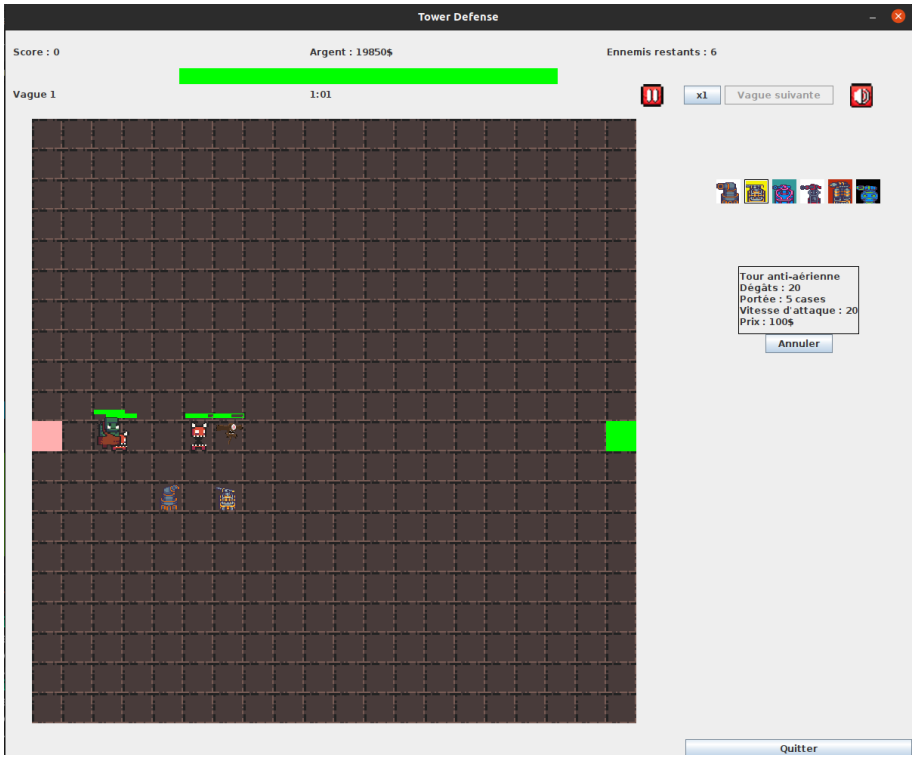
1.1 Notre sujet

Pour notre projet de programmation du semestre 4, nous avons opté pour le sujet *Tour de Défense*, proposé par Mme Bérénice DELCROIX-OGER.
Le but de ce projet était de créer un jeu du type tour de défense en 2D en langage

Java à l'aide de ce que nous avions vu jusqu'au semestre 3 en programmation, essentiellement en programmation orientée objet dans notre cas.

1.2 Présentation du produit

Tout le long du semestre nous avons donc développé *Tower Defense* pour répondre au sujet :
il s'agit donc un jeu de type tour de défense 2D, situé dans un univers fantasy, dans lequel le joueur doit poser différents types de tours qu'il achète avec son argent, afin d'éliminer les ennemis apparaissant sur la map (sur la case rose) avant qu'ils n'atteignent leur objectif (la case verte). Chaque ennemi rapporte de l'argent et certains ne peuvent être atteints que par certains types de tours (par exemple : les chauves-souris ne peuvent être battues qu'à l'aide des tours anti-aériennes).



Capture d'écran provenant du jeu

Il existe de nombreuses stratégies possibles, il est possible à l'aide du calcul de la trajectoire par l'algorithme A* implémenté (que nous développerons plus tard) de "contrôler" la trajectoire des ennemis, comme illustré grâce à la capture d'écran ci-dessus.

2 Structure du projet

Ce projet utilise la structure MVC (modèle, vue, contrôleur) étudiée en POO.

2.1 Diagramme de classe

2.1.1 Modèle

Nous avons donc établi le modèle de manière classique en ce qui concerne la gestion de la partie et du tableau avec les classes Board, Game, Player, Projectile, Score, Tile et Wave, qui gèrent bien séparément tout ce qu'elles ont à faire tout le long de la partie sur l'évolution du jeu en lui-même.
Pour les ennemis et les tours, nous avons créé les classes Enemy et Tower comme bases pour pouvoir, par héritage, créer facilement les différents types de tours (avec les classes AerialTower, BasicTower, DestructiveTower, InfernalTower, RapidTower, SuperTower) et d'ennemis (avec les classes AerialEnemy, BasicEnemy, Mo et TankEnemy) s'il nous venait de nouvelles idées (ou simplement en cas de besoin).
Les fichiers correspondant aux tours et aux ennemis sont, par souci de clarté, regroupés dans dossiers respectifs tower et enemy.

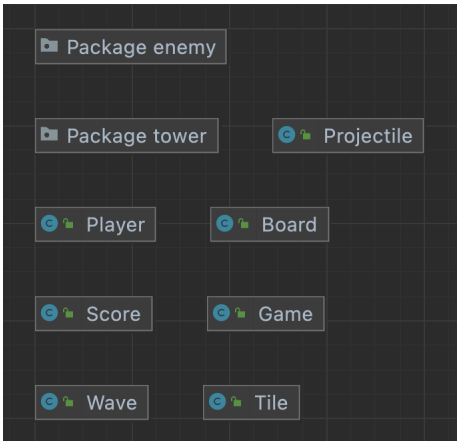


Diagramme de classes correspondant au modèle

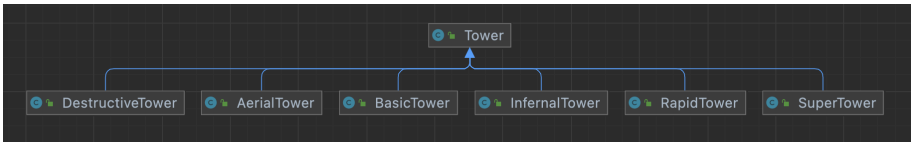


Diagramme de classes correspondant au package tower

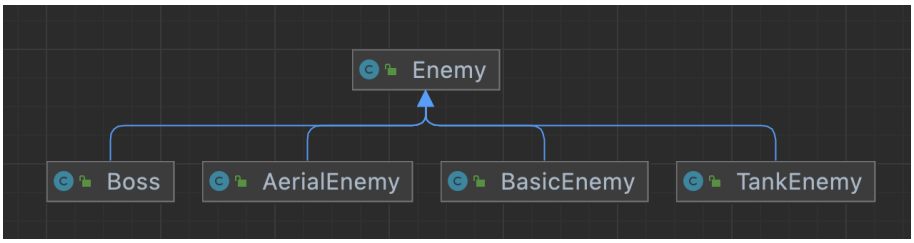


Diagramme de classes correspondant au package enemy

2.1.2 Vue

La vue quand à elle se compose d'un dossier contenant les classes qui gèrent séparément (pour un souci de modularité) les différents éléments relatifs au menu, soit les classes Accueil, Discover, EndGame, HighScore, HighScoreInMenu, ImagePanel, Menu, MyButton, NewGame, NewPlayer, Paramètres et SoundManager.

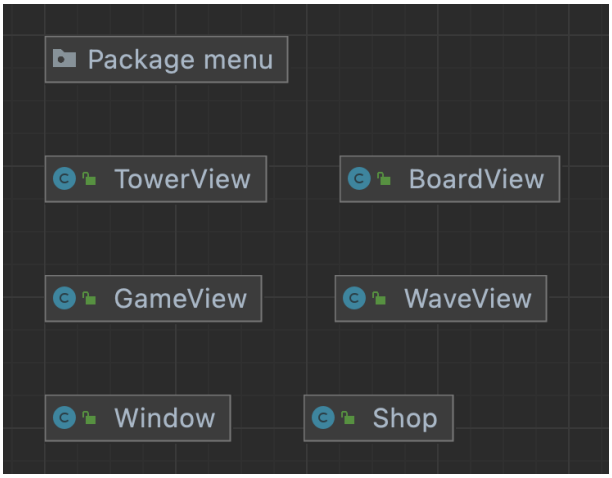


Diagramme de classes correspondant à la vue

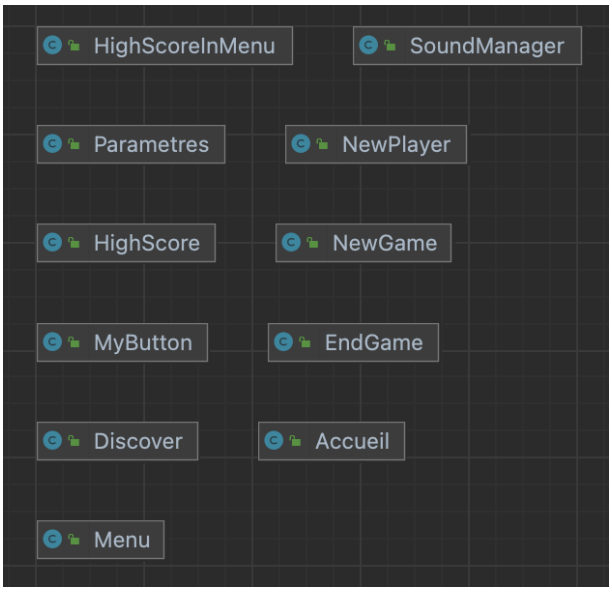


Diagramme de classes correspondant au package menu

2.1.3 Contrôleur

Le contrôleur ne se compose que de la classe Launcher qui se contente de créer une nouvelle fenêtre dans le main.

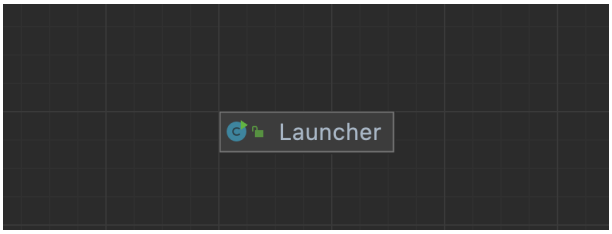


Diagramme de classes correspondant au contrôleur

2.2 Organisation et répartition des tâches

Pour répartir les tâches, nous n'avions pas de plan d'attaque particulier, nous nous sommes contentés de lister chaque semaine les tâches que nous voyions puis de se les répartir à peu près uniformément pour les réaliser sur la semaine puis s'entraîder en cas de besoin.

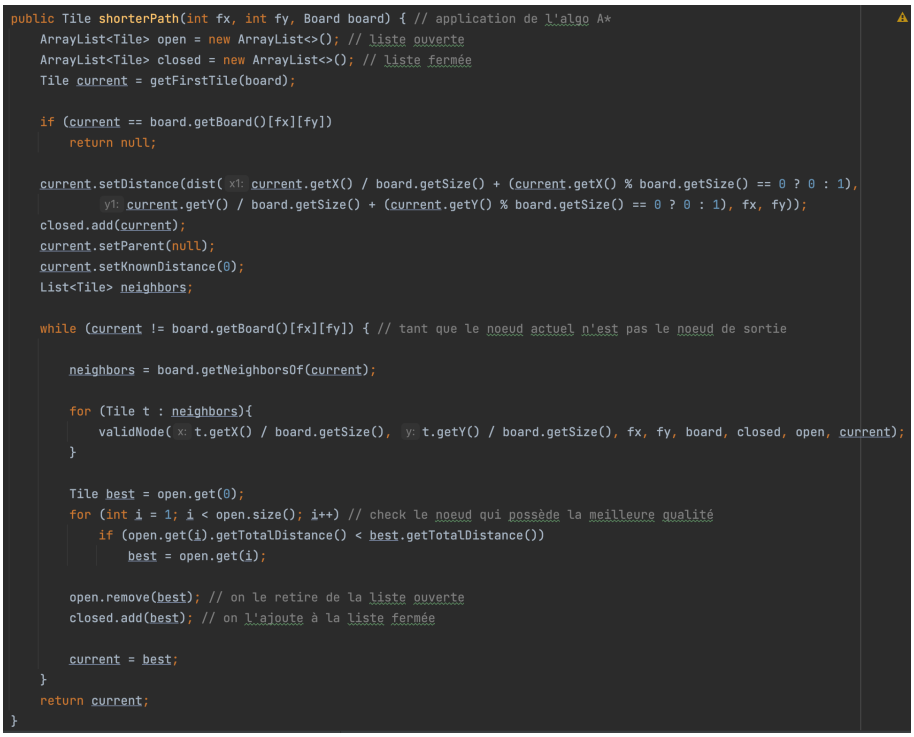
Nous avons au départ utilisé les différentes fonctionnalités de GitLab pour communiquer, puis nous avons réalisé que nous arrivions au final à bien se concerter directement lors des réunions hebdomadaires et que dans le cas où un membre

avait besoin de précisions ou d'aide pour sa tâche, nous pouvions demander plus facilement par d'autres moyens de communication.

3 Difficultés spécifiques et générales rencontrées

3.1 Le déplacement des ennemis

Une des difficultés spécifiques majeures rencontrées portait sur le déplacement des ennemis, et plus précisement sur le calcul de la trajectoire et du plus court chemin entre le point de départ et le point d'arrivée.
Pour palier à ce problème, nous nous sommes d'abord renseignés sur l'algorithme de Dijkstra et l'algorithme A*, avant de pencher pour l'algorithme A* pour des raisons d'efficacité :
Dans la classe Enemy.java, nous définissons globalement les caractéristiques de nos ennemis : le montant de leurs points de vie actuels et de base, leurs coordonnées sur le plateau, la direction qu'ils doivent prendre (haut,haut-gauche,haut-droite,gauche,droite,bas,bas-gauche,bas-droite) sous forme d'un int et le chemin qu'ils doivent prendre sous forme d'une pile de Tile (qui sont les cases du plateau).
Les déplacements des ennemis sont calculés à l'aide de l'algorithme A*. Le principe est de regarder quel chemin réduit le plus la distance entre les coordonnées de départ et celles d'arrivées.
Pour calculer la distance, nous avons choisi la formule de la distance entre deux points basée sur la norme euclidienne du plan.
Nous avons deux listes : une ouverte et une fermée. La liste ouverte contient les cases qu'on peut possiblement emprunter et la liste fermée contient les cases qu'on parcoure réellement.
On regarde les cases voisines de l'ennemi : si la case est accessible (donc ne contient pas de tour ou est déjà présente dans la liste fermée) on l'ajoute dans la liste ouverte. Puis dans la liste ouverte on regarde quelle case possède la plus petite distance par rapport à l'arrivée, cette case sera ajoutée dans la liste fermée. Chaque case possède une case parent qui permet de retracer le chemin que parcoure l'ennemi, c'est pour cela que c'est stocké dans une pile.
Les différents types d'ennemis sont également modélisés dans les sous-classes de Enemy.java : BasicEnemy, AerialEnemy, TankEnemy, etc.



```
public Tile shorterPath(int fx, int fy, Board board) { // application de l'algo A*
    ArrayList<Tile> open = new ArrayList<>(); // liste ouverte
    ArrayList<Tile> closed = new ArrayList<>(); // liste fermée
    Tile current = getFirstTile(board);

    if (current == board.getBoard()[fx][fy])
        return null;

    current.setDistance(dist( x1: current.getX() / board.getSize() + (current.getX() % board.getSize() == 0 ? 0 : 1),
            y1: current.getY() / board.getSize() + (current.getY() % board.getSize() == 0 ? 0 : 1), fx, fy));
    closed.add(current);
    current.setParent(null);
    current.setKnownDistance(0);
    List<Tile> neighbors;

    while (current != board.getBoard()[fx][fy]) { // tant que le noeud actuel n'est pas le noeud de sortie

        neighbors = board.getNeighborsOf(current);

        for (Tile t : neighbors){
            validNode( x: t.getX() / board.getSize(),  y: t.getY() / board.getSize(), fx, fy, board, closed, open, current);
        }

        Tile best = open.get(0);
        for (int i = 1; i < open.size(); i++) // check le noeud qui possède la meilleure qualité
            if (open.get(i).getTotalDistance() < best.getTotalDistance())
                best = open.get(i);

        open.remove(best); // on le retire de la liste ouverte
        closed.add(best); // on l'ajoute à la liste fermée

        current = best;
    }
    return current;
}
```

Capture d'écran de l'implémentation de l'algorithme A*

3.2 Visualisation du plateau

Assez tôt dans la conception du projet, nous avons décidé de donner aux ennemis leurs coordonnées en pixels en fonction de leur position sur le plateau. Nous avons rencontré un problème pour l'exécution de l'algorithme qui permet aux ennemis de se déplacer vers la sortie en prenant le chemin le plus court. Nous avons donc choisi d'utiliser un tableau de cases en deux dimensions. Nous avons créé un attribut 'size' (qui vaut 40) et chaque case du tableau fait 'size' pixels. La case [0, 0] est donc représentée aux coordonnées [0, 0], la case [1, 1] est aux coordonnées ['size', 'size'], la case [3, 6] aux coordonnées [3 * 'size', 6 * 'size'], etc. Nous calculons la position des ennemis sur le plateau en prenant leur position en pixels divisé par 'size'.

3.3 Tours et projectiles

Tower.java est la classe qui définit toutes les caractéristiques générales d'une tour. Elle définit sa position sur le plateau, quel ennemi elle doit attaquer, ou encore elle lui permet d'attaquer cet ennemi. Les différentes classes qui héritent de Tower.java permettent quant à elles de définir les caractéristiques spécifiques à chacune des différentes tours : par exemple le montant de dégâts infligés, le prix , la vitesse d'attaque ou le type d'ennemis pouvant être ciblés figurent dans

ces classes spécifiques.
Une fois qu'un ennemi est à portée de tir, la tour attaque. Elle créé un objet de type Projectile en fonction de sa vitesse d'attaque. Les projectiles ont comme attributs des coordonnées et un ennemi. Pour que le projectile se déplace vers l'ennemi, on calcule l'angle de la droite qui passe par sa position et la position de l'ennemi. Cela permet au projectile de suivre un ennemi alors que celui-ci se déplace également.

3.4 Système de score

Nous avons pensé que le jeu serait plus amusant si à la fin d'une partie nous n'avions pas seulement gagné ou perdu.
Nous avons donc mis en place un sytème de score qui permet au joueur de relancer une partie en essayant de battre son score précédent. Le score est calculé en fonction des ennemis tués et de la vitesse à laquelle on enchaîne les vagues. À la fin d'une partie, il est possible d'enregistrer son score en entrant un nom d'utilisateur. La classe Highscore.java gère les meilleurs scores enregistrés : pour cela, elle ouvre un fichier highscore.txt et y inscrit les différents scores.
A la création d'un objet de type HighScore, on vérifie que le fichier .txt existe. Si c'est le cas, on lit son contenu et le stocke dans une liste de Score, sinon on crée un nouveau fichier vierge. Pour ajouter un nouveau score, on l'insère dans la liste de manière que la liste reste triée en ordre décroissant. Ensuite, on remplace le contenu du fichier .txt par le contenu de la liste.
Si la liste contient plus de dix scores, on garde uniquement les dix plus grands.

3.5 Le système de vagues

Les vagues possèdent comme attribut la durée de la vague défini par un Timer, le nombre d'ennemis par vague défini par un int, la vague courante et le nombre max de vagues définis par des int.
Cette classe a également accès au plateau et au joueur via l'attribut Game game.
Chacune des vagues est gérée par un switch() jusqu'à la dernière vague du mode difficile, puis les vagues supérieurs du mode infini sont automatisées.
Dans chaque case du switch, la fréquence d'apparition et les points de vie des ennemis sont gérés en fonction de la difficulté et le nombre d'ennemis augmente, pour avoir un certain équilibre dans le jeu.

4 Liens et ressources

Pour voir le code de notre projet en détail, cliquez ici ou rendez vous sur la page `https://gaufre.informatique.univ-paris-diderot.fr/debbah/tower-defense`.