Rapport de Projet de POO

Dikesh Uprety et Ilan Dahmane

Le projet vise à concevoir et développer un jeu de tower defense, un genre de jeu stratégique où le joueur doit défendre une zone contre des vagues d'ennemis en plaçant des tours de défense.On a choisi le style Plant vs Zombie.

Pour satisfaire la conception de jeu, on a utilisé le logiciel de gestion de versions GIT(GITLAB).

Les outils

Logiciels.

Pour réaliser ce projet , on a dû utiliser visual studio code et extension live share. Ce dernier nous a permis de travailler ensemble en temps réel.
On a également utilisé le logiciel d'appel, Discord. Qui nous a permis de parler et partager nos écrans. Et bien sûr la VCS , Gitlab.

Les Ressources

Notre binôme a utilisé plein des ressources, afin de créer de bons programmes.
On s'en sert des documentations officielles des java. Pour mieux comprendre les méthodes natives. Autre ressource que on a utilise est: https://waytolearnx.com/
On a également profité des vidéos youtube.

Les phases de développement de Projet

Explication des besoins

Dans cette phase on a discuté sur quel type de tower defense. On a choisi le style Plant vs zombie. On a également soigneusement lu toutes les contraintes à respecter présentes dans le cahier de charges .

Conception

On a eu une discussion sur l'architecture de notre projet. C'est à dire class et les packages que on va créer. Dans cette phase on a pu créer notre première brouillon entre les liaisons

des classes. Cette vue des classes a dû se développer au fur à mesure. Enfin à être de plus en plus concret.

Implementation

On a choisi en première d'implémenter Le mode console. Ce choix se justifie par le fait que d'après le consigne de PDF. Mais également au moment de fait, on avait Zéro connaissance sur interface graphique donc sur swing et awt.

cahier des charge

Tous les composants essentiels énoncés dans le cahier des charges minimal ont été intégrés dans la version terminal de notre projet.

Nous avons développé plusieurs niveaux au sein de cette version, notamment les niveaux Facile, Moyen et Difficile. Chaque niveau se caractérise par des mécanismes distincts de spawn des ennemis, offrant ainsi une diversité en termes de complexité et de challenge pour l'utilisateur.

Par ailleurs, nous avons enrichi l'expérience de jeu en introduisant différents modes de jeu au sein de la version terminal. Plus spécifiquement, nous avons mis en œuvre les modes "Normal" et "Marathon", apportant ainsi une variété de configurations et de défis pour l'utilisateur.

En ce qui concerne la variabilité du contenu, nous avons diversifié le tableau de jeu en intégrant une gamme de zombies aux caractéristiques et comportements variés, contribuant ainsi à l'enrichissement de l'expérience de jeu.

Concernant l'interface graphique, nous avons adopté une approche minimaliste en utilisant exclusivement les bibliothèques Swing et AWT, garantissant ainsi une intégration cohérente et une performance optimale, tout en respectant les contraintes et les spécifications du projet.

les pistes d’extensions

En raison de contraintes temporelles et techniques, nous n'avons pas été en mesure de finaliser la version graphique de notre jeu à la hauteur de nos aspirations initiales. Il était envisageable d'incorporer une diversité de niveaux supplémentaires au sein de l'interface graphique, en complément de l'ajout d'un mode "Marathon" pour enrichir l'expérience utilisateur. Par ailleurs, une piste d'amélioration substantielle aurait été la mise en place d'une fonctionnalité de sauvegarde du jeu, permettant ainsi aux utilisateurs de conserver leur progression et de reprendre l'aventure à leur convenance.

Architecture de projet

On a utilisé l'architecture model et vu. Notre projet Contient 3 Modules principaux, **model** et **vu et controlleur** . Dans le module **model.** On a géré toutes les logiques de jeu et l’affichage de jeu en terminal. Le modèle continent les sous modules: **Defender** et **Enemy.**
Dans model:

- **Mob** : c’est une class abstract.la classe Mob sert de fondation pour la modélisation des entités mobiles dans votre jeu. Elle encapsule des propriétés essentielles et des comportements communs, tout en offrant une structure extensible pour la définition de types de mobs spécifiques à travers des classes dérivées.

- **Mapconfig** : Représente une implémentation détaillée d'un jeu de stratégie où le joueur doit utiliser des défenseurs pour protéger sa base contre des vagues d'ennemis. Il combine plusieurs aspects, tels que la génération aléatoire des ennemis, la gestion des ressources du joueur, et la logique de déplacement et d'attaque pour créer une expérience de jeu complexe et engageante.
    - **Structure de base**: La classe MapConfig gère les configurations de la carte, y compris la création de la carte elle-même, l'initialisation des défenses et des ennemis, et la logique de déplacement et d'attaque.
    - **Configuration de la carte**: on a deux constructeurs, l'un pour le mode terminal et l'autre pour l'interface graphique. Chacun de ces constructeurs initialise la carte avec différentes configurations basées sur le mode de jeu choisi.
    - **Méthodes de configuration de difficulté**: Les méthodes pour_facile(), pour_Moyen(), pour_dur(), et Pour_Marathon() sont destinées à configurer les paramètres pour différents niveaux de difficulté ou modes de jeu. Chaque méthode initialise la carte avec des paramètres spécifiques.

- **Initialisation de la carte**: La méthode basic() initialise les éléments de base de la carte, y compris le joueur et la structure de la carte elle-même.
- **Gestion des ennemis**: La méthode spawn() est responsable de la génération des ennemis en fonction du mode de jeu sélectionné. Elle utilise un générateur aléatoire pour déterminer les types et les emplacements des ennemis sur la carte.
- **Déplacement et attaque**: Les méthodes move_Left() et attack() gèrent respectivement le déplacement des ennemis et l'attaque des défenseurs. Ces méthodes sont essentielles pour la dynamique du jeu, où les défenseurs doivent éliminer les ennemis avant qu'ils n'atteignent la base.
- **Autres fonctionnalités**: Le code comprend également des méthodes pour afficher l'état actuel de la carte, ajouter des défenses, vérifier les conditions de victoire ou de défaite, et d'autres fonctionnalités liées à la logique de jeu

- **Player** : cette classe offre une structure simple pour représenter un joueur avec des attributs d'argent, de score, et de nombre de tués, ainsi que des méthodes pour accéder et modifier ces attributs.
- **launcher** : la classe sert de point d'entrée pour votre application de jeu. Elle gère les entrées de l'utilisateur, permet de sélectionner un niveau de difficulté et initialise le jeu avec les paramètres choisis.

A l'intérieur de module mob, on a les modules Enemy et Defender.

Dans Module Defender

1. Classe Defender :

- Cette classe hérite de Tower.
- Elle définit une tour avec un nom "Defender" et un affichage "C".
- La méthode get_display() affiche l'état de la tour en fonction de ses points de santé.
- Le prix de cette tour est fixé à 100.

2. Classe Defender1 :

- Encore une fois, cette classe hérite de Tower.
- Cette tour est nommée "Defender1" avec un affichage "N".
- La méthode get_display() est similaire à celle de Defender.
- Le prix de cette tour est de 80.

3. Classe Defender2 :

- Hérite également de Tower.
- Représente une tour "Bombe" avec un affichage "B".
- La méthode get_display() affiche l'état de la tour et peut également attaquer des ennemis adjacents (selon la logique d'attaque définie).
- Cette tour peut effectuer une attaque particulière avec la méthode attack(MapConfig battle).
- Le prix de cette tour est également de 80.

4. Classe Tower :

- Cette classe est la classe mère pour toutes les tours et hérite de Mob.
- Elle a un constructeur qui initialise le nom, l'affichage et l'intensité des dégâts.
- La méthode attack(MapConfig battle) définit la logique d'attaque de base pour toutes les tours.
- Il y a des méthodes pour vérifier la présence d'ennemis dans des positions spécifiques (getEnemyPresent et getEnemyPresent_column).

Dans Module Enemy

1. Classe abstraite Enemy:

- Cette classe abstraite hérite de Mob.
- Elle contient une logique pour afficher l'état de l'ennemi avec la méthode get_display(). L'affichage varie en fonction des points de santé de l'ennemi.

2. Classe Enemy1 :

- Cette classe hérite de Mechant.
- Elle représente un type spécifique d'ennemi, un zombie avec des attributs et des comportements spécifiques.
- Le constructeur initialise divers attributs, le nombre de frames, les points de santé, etc.

3. Classe Enemy2 :

- Cette classe hérite également de Mechant.
- Elle représente un autre type d'ennemi avec ses propres spécificités.

4. Classe Mechant :

- Cette classe hérite de Mob et représente un type d'ennemi.
- Elle a des attributs pour suivre l'état de l'ennemi, tels que s'il est mort, s'il marche, s'il attaque, etc.
- La méthode attack (MapConfig battle) définit le comportement d'attaque de base pour un ennemi, réduisant les points de santé d'une cible sur la carte.

Dans la module View

1. **plantvsZombie**

Cette classe représente les mécanismes de jeu principaux et le rendu de l'interface utilisateur graphique pour un jeu de style "Plants vs. Zombies".

2 Start

constitue l'écran d'accueil de jeu, offrant des options pour lancer différents modes ou fonctionnalités.

3 Levels

sert de menu interactif où les utilisateurs peuvent choisir le niveau de difficulté d'un jeu en cliquant sur les boutons appropriés. Une fois un niveau sélectionné, la fenêtre se ferme et lance une nouvelle instance du jeu avec le niveau de difficulté choisi.

4 Card

Card est conçue pour représenter et manipuler des cartes graphiques dans une interface utilisateur. Elle permet d'afficher des cartes qui peuvent être achetées ou non, et elle fournit également une méthode pour vérifier si une carte spécifique a été "pressée" ou sélectionnée en fonction de coordonnées fournies.

Dans module Controlleur

1) **Mouse :** cette classe gère les interactions de l'utilisateur avec l'interface graphique du jeu pour acheter et placer des défenseurs sur le terrain.

Comment lancer le jeu?

Vous pouvez lancer le jeu depuis Start.java depuis un IDE .

Les problèmes qu' on a rencontrés durant le projet.

C'était un projet assez compliqué pour nous. C’était la première fois que nous pouvions créer un programme autant densément orienté objet. Également la première fois qu' on fait un programme graphique .

Nous avons rencontré d'importants obstacles relatifs à la gestion des animations dans l'environnement Swing, particulièrement en ce qui concerne la manipulation des images GIF. Initialement, nous avons tenté d'aborder cette problématique en affichant les images individuellement à l'aide d'un indice courant, que nous incrémentant à intervalles réguliers définis par un timer. Néanmoins, cette approche s'est avérée peu pratique en raison de la complexité croissante introduite par la nécessité de gérer plusieurs animations distinctes (telles que la mort, l'attaque, la marche) avec des nombres de frames variables. Cette méthode s'est avérée non seulement laborieuse à mettre en œuvre mais également difficile à maintenir.

Par la suite, nous avons opté pour une solution hybride, combinant l'utilisation d'images individuelles et de GIFs, afin de pallier les limitations observées avec les GIFs, notamment la présence résiduelle d'artefacts indésirables tels que des carrés noirs.

Un autre défi majeur résidait dans la synchronisation des pixels entre les éléments graphiques et le tableau de configuration de notre carte, représenté par un tableau bidimensionnel Case [] []. Il a été impératif d'adopter une approche plus calculatoire et rigoureuse pour gérer ces aspects, incluant le suivi du curseur de la souris et l'adaptation dynamique de l'interface en fonction des dimensions de l'écran.

Notre décision initiale de privilégier une approche frame par frame a été remise en question le mercredi soir, en raison des défis inhérents à la gestion des timers et des indices courants.

En outre, nous avons rencontré des difficultés significatives dans la gestion des entités mobiles (mobs) sur la carte. L'adaptation de leurs animations en fonction des différentes situations s'est avérée particulièrement ardue, notamment en ce qui concerne la gestion des collisions et du mouvement. De plus, la gestion du déplacement des mobs s'est compliquée en raison de contraintes liées à la manipulation des variables d'axe x partagées entre les différents mobs, ce qui a conduit à des comportements imprévisibles et des bugs.

Dans le cadre de notre implémentation, nous avons intégré la logique de déplacement des ennemis (Enemy) au sein du tableau mapConfig. Lors de la mise en œuvre de cette fonctionnalité, nous avons rencontré des défis particuliers liés à la gestion dynamique des indices de colonne (j) pour les ennemis en déplacement.

Initialement, notre approche pour déplacer les ennemis consistait en une vérification conditionnelle, telle que présentée ci-dessous :

```
if (map[i][j].Get_Enemy_present()) {
    posiX = ((j * cellWidth) - cellWidth) + gifX;
    // ... Logique de déplacement des ennemis
}
```

Cependant, cette méthode a engendré des incohérences et des imprévisibilités en raison de la modification continue de la variable j pendant le déplacement des ennemis dans le tableau. Pour pallier ce problème, nous avons introduit une stratégie de gestion plus rigoureuse.

Concrètement, nous avons incorporé une variable supplémentaire au sein des classes dédiées aux ennemis. Cette variable, que nous avons nommée colonnePosi , a été conçue pour conserver la valeur initiale de la colonne (j) où l'ennemi apparaît initialement dans le tableau.

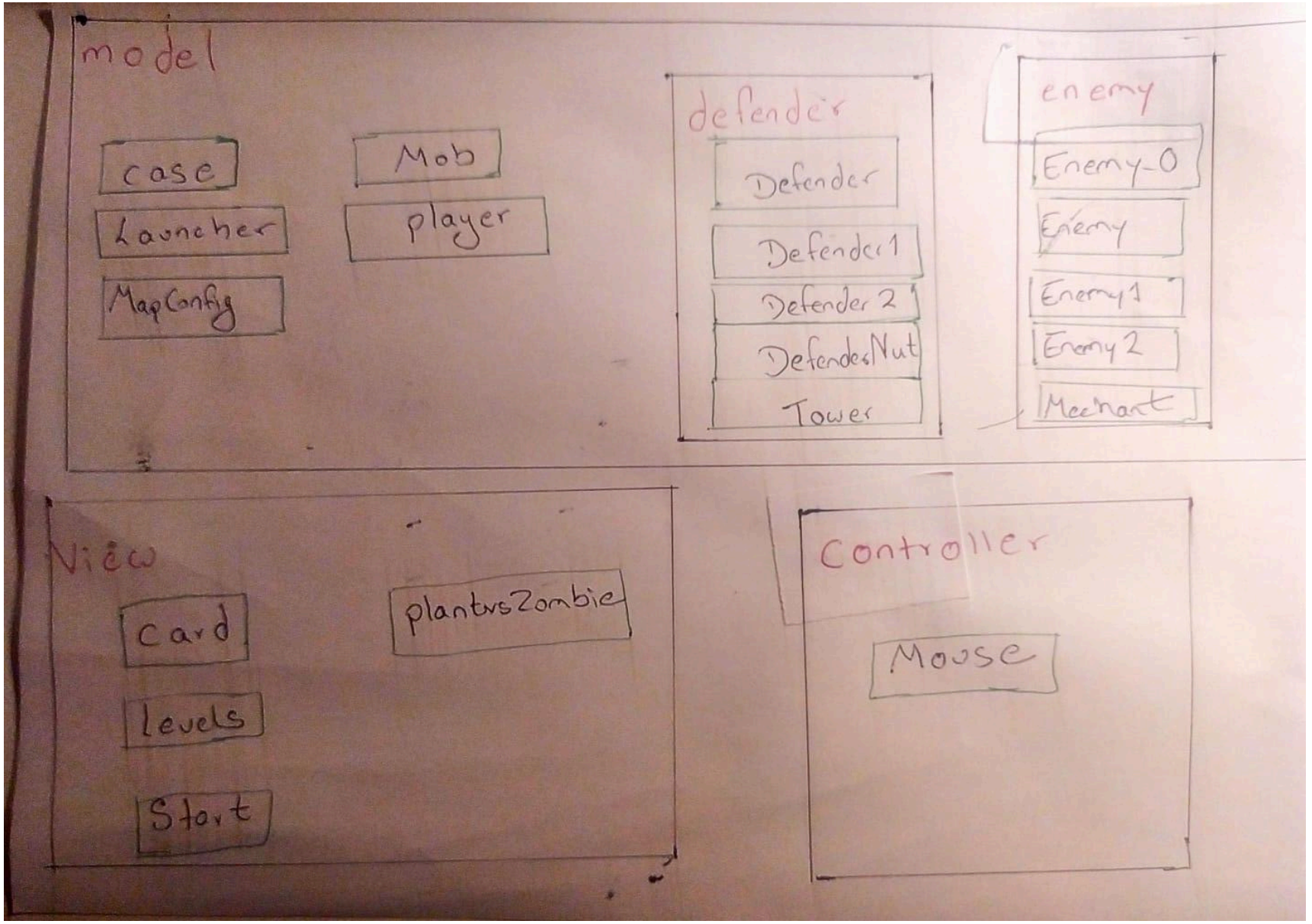
En conséquence, lors de l'application de la logique de déplacement, nous avons ajusté le calcul de la position horizontale (posiX) de la manière suivante :
posiX = ((currentMob.getColonnePosi() * cellWidth) - cellWidth) + gifX;
En adoptant cette approche, nous avons pu garantir une cohérence et une prévisibilité dans le déplacement des ennemis, en s'assurant que les calculs de positionnement s'exécutent toujours à partir de la valeur initiale de la colonne (colonnePosi) associée à chaque ennemi. Ainsi, nous avons éliminé les problèmes potentiels d'affichage erroné ou d'avancement inattendu des ennemis au sein de la carte.

En somme, la mise en œuvre d'un système de jeu robuste et modulaire s'est avérée être un défi considérable, nécessitant une réévaluation constante de nos approches et une résolution proactive des problèmes rencontrés.

Conclusion

Même si le projet n'est pas terminé, il a été une expérience précieuse. Il nous a permis d'apprendre et de mettre en pratique de nombreuses compétences en programmation, conception logicielle et gestion de projet. Malgré les défis rencontrés, cette expérience nous a enrichis et nous a aidés à progresser dans nos études et compétences professionnelles.

Cependant, il serait grandement apprécié si vous pouviez nous présenter une méthodologie structurée pour la création d'un jeu ou d'un projet. Une telle guidance nous permettrait de mieux comprendre les étapes nécessaires, les meilleures pratiques et les stratégies efficaces pour mener à bien notre initiative.


model
case
Launcher
MapConfig
Mob
player
defender
Defender
Defender1
Defender2
DefenderNut
Tower
enemy
Enemy-0
Enemy
Enemy1
Enemy2
Mecmart
View
card
levels
Start
PlantvsZombie
Controller
Mouse