Compte-Rendu Projet POOIG3 Sekou Cisse et Dimitri Fernez

Sommaire :

1) Introduction :

À la demande des responsables de l'UE de POOIG3, il nous étaient demandé de réaliser un projet de type Tower Defense en Java respectant soit le modèle de Bloons TD ou de Plants vs Zombie. Nous avons décidé de suivre le modèle de Plants vs Zombie pour des raisons esthétiques et d'envie et nous avions jusqu'au 11 Janvier 2024 pour rendre le projet à temps et fini.

1.1) But du projet :

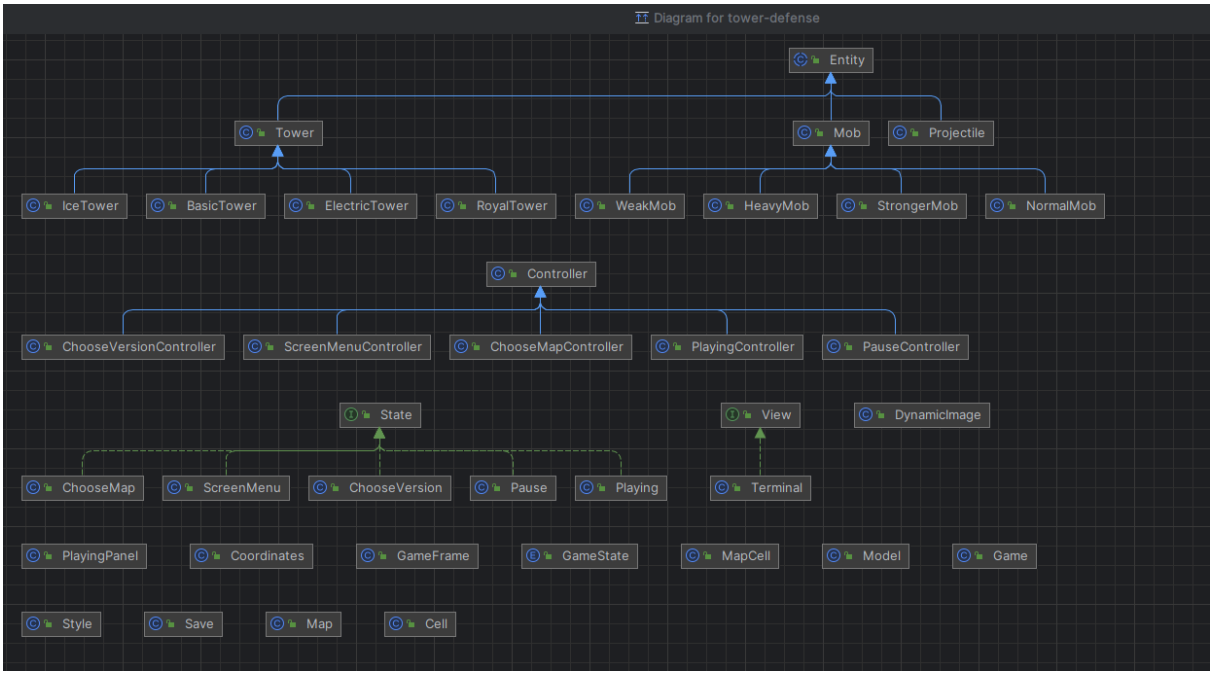
Le but du projet étaient de faire un Plants vs Zombie mais dans un monde fantastique où l'on retrouve multiples animaux (tel que des ogres, des abeilles, des chats et des slimes) ayant pour but d'anéantir les humains à la demande de mère nature. Donc le but est d'opposer des archers aux animaux hostiles et de rendre tout cela amusant.

1.2) Nos attentes à la fin du projet :

- Pouvoir réaliser des animations entre chaque scène pour donner l'air au jeu d'être plus fluide.
- Pouvoir animer le mouvement des entités et avoir une animation d'attaque et de marche pour chaque entité (Animaux ou Personnes).
- Pouvoir avoir au moins 2 maps différentes.
- Avoir un jeu devenant plus difficile à la demande et aussi au fil du temps.
- Avoir une bonne connaissance de Swing et AWT pour le prochain semestre.
- Avoir une bonne maitrise de Git.
- Apprendre à travailler en équipe et trouver des compromis lors de conflits.

2) Fonctionnement :

Diagramme des classes :



2.1) Les entités :

> Les **entités** représente toutes personnes pouvant être placé dans le plateau. On en différencie 3 représentant ces seuls sous-classes :
>
> - Les `Mobs`, les animaux hostiles voulant attaquer les humains;
> - Les `Towers`, les humains essayant de se défendre;
> - Les `Projectiles`, les armes utilisés par les *towers* pour attaquer les mobs.

Son importance est cruciale pour le fonctionnement des jeux. Étant donné que la classe `Entity` est importante et qu'elle sera différence en fonction de la personne, nous avons décider de la rendre abstraite pour que ces trois *entités* aient un comportement différent en fonction de la classe mais voulant exprimer la même chose.

C'est aussi dans cette classe qu'on retrouve la fonction qui charge l'image de chaque entité. Chaque sous-classe d'`Entity` ont différents points en commun :

- Ils ont 4 **Strings** important représentant chacun :
  - Le chemin vers l'image courant de l'*entité (`currentImage`);*
  - Le chemin vers l'image représentant l'*entité* en état normal (`entityWalk`);
  - Le chemin vers l'image représentant l'*entité* en mode attaque (`entityAttack`);
  - Le chemin vers l'image représentant l'*entité* en mode mort (`entityDead`)
- Un **boolean** permettant de savoir si l'entité se trouve à l'écran
- Plusieurs informations utiles tels que le nom de l'entité, son printTerminal(voir Mode Terminal), etc
- Un **Timer** pour les Mobs et les Tours pour attaquer (Voir Les Mobs et Les Tours)
- Les **Coordonnées** utiles pour le placement de l'*entité* dans la Map et dans l'interface graphique (Voir Les Coordinates);

2.1.1) Les Mobs :

> La classe **Mob** représente les entités ayant pour but de franchir tout le chemin en tuant toutes les tours se trouvant dans le chemin. Elle fait perdre le joueur à partir du moment où elle franchi complètement sa ligne courante. Elle est représenter par 4 sous-classe :
>
> - Les `WeakMob`, représentant les ennemis les plus faible de la classe Mob. Il est très facile à tuer. Ils sont générés dès la 1ère vagues.
> - Les `NormalMob` sont des ennemis étant assez facile à tuer. Mais à partir du moment où les *NormalMob* attaquent une tour, leur élimination deviennent assez compliqué. Ils sont générés dès la 3ème vagues.
> - Les `HeavyMob` sont des ennemis étant fort. Lorsqu'ils commencent à apparaître, le jeu commence à devenir plus dur et il est nécessaire d'avoir des tours d'au moins niveau 3 pour pouvoir les tuer efficacement. Ils sont générés à partir de la 5ème vagues.
> - Les `StrongerMob` sont les ennemis infligeant le plus de dégâts. Il est presque impossible de les tuer avec seulement une seul tour de niveau 1 et il faut nécessairement plusieurs tours de niveau 2 ou 3. Ils sont générés à partir de la 7ème vagues.

Comme pour les *towers*, elle est l'une des classes de `Entity` à utiliser un Timer pour pouvoir attaquer une tour. Cela permet d'avoir un intervalle constant entre chaque attaque. Ce timer sera mis à jour à chaque fois que la fonction `attack()` est appelé.

Chaque *Mob* contient le même `printTerminal` (correspondant au caractère ' X ') qui permet de différencier les entités entre eux et les sous-classes sont différenciés grâce à une couleur donné.

2.2.2) Les Tours :

> La classe **Tower** représente les entités ayant pour but d'empêcher les *Mobs* de franchir la ligne et de les tuer en conséquence. Elle est l'interaction directe avec le joueur qui le place directement dans le plateau de jeu. Elle est représenter par 4 sous-classe :
>
> - Les `BasicTower`, sont les tours infligeant le moins de dégâts. Leur avantage étant qu'elle sont celle qui tire le plus rapidement mais aussi celle qui sont le plus vulnérable. Elles sont assez efficace face au `WeakMob` mais seulement s'ils sont très peu.
> - Les `NoviceTower`, représente la version amélioré de la `BasicTower`. Elle reste plus efficace mais aura du mal face aux ennemis nombreux. Elle est efficace pour retenir les slimes pendant un moment.
> - Les `VeteranTower`, sont les tours les plus avantageux au niveau du prix. Elles représente une forte amélioration des `BasicTower` et des `NoviceTower` et elle est efficace pour tenir les mobs hors de portée pour un long moment.
> - Les `RoyalTower`, sont les meilleurs tours possible. Elles infligent le plus de dégâts et arrivent à tenir face au mobs les plus fort. Elle est aussi la tour la plus chère et elle reste très efficace pour éliminer les mobs rapidement.

Comme pour les *mobs,* elle est l'autre classe de `Entity` à utiliser un Timer pour pouvoir attaquer un mob. Le Timer se chargera de créer de nouveau `Projectile` et de les placer dans la map grâce à une liste contenant tous les projectiles créé.

Chaque *Tower* contient le même `printTerminal` (correspondant au caractère ' T ') qui permet de différencier les *entités* entre eux et les sous-classes sont différenciés grâce à une couleur donné.

Un des problèmes rencontré étaient que les *towers* ne détectaient pas les *mobs* quand elles étaient dans les mêmes coordonnées. Donc pour cela, nous avons créer une méthode qui parcourt les listes des Mobs et attaque s'il y'en a bien un dans la même coordonnée.

Aussi, un autre problème rencontré étaient le fait que les *Towers* tiraient rapidement lorsque les *Mobs* étaient collé au tour, et vice-versa. C'est pour cela qu'on a implémenter des *Timers* pour pouvoir gérer leur temps d'attaque et en plus, cela rajoute de la difficulté au jeu.

2.2.3) Les Projectiles :

> La classe **Projectile** représente les entités, générés par une *tower*, permettant d'attaquer le mob le plus proche se trouvant dans sa trajectoire. Elle est la classe à nous avoir causé le plus de problème.

Elle se différencie de `Tower` et `Entité` car elle est représenter par une seule classe et elle n'a pas vocation à être utile pour un long moment. Elle se chargera d'infliger les dégâts au *mob* touché et de s'occuper de sa mort si besoin.

Chaque `Projectile` contient le même `printTerminal` (correspondant au caractère ' . ') qui permet de différencier les *projectiles* entre eux. Ils prendront la couleur de leur *tower* parent.

Le problème avec l'implémentation actuelle de notre projet, c'est que les *projectiles* étaient plus rapide que certains mobs et donc n'arrivait pas à les voir dans leur emplacement.

Un autre problème aussi était qu'un fusionnait avec d'autre project lorsque ce projectile étaient derrière un autre projectile et qu'il était traité en premier. Pour cela nous avons trié les projectiles de sorte à ce que le projectile le plus devant soit traité en premier.

2.2) Le mode Terminal :

> Le **mode Terminal** est la version du jeu permettant de jouer "textuellement" au jeu. Bien qu'elle soit un peu différent de la

> version graphique, elles partagent le même modèle et donc peuvent être lancé simultanément.

Bien qu'elle n'est pas obligatoire, le *mode Terminal* était d'une importance crucial pour le projet. Ayant préféré commencer par le mode terminal, il serait plus simple de debug le projets rapidement et permettra une implémentation facile du mode graphique une fois terminé.

2.1.1) L'utilisation des Scanners :

> Les **Scanners** sont des objets permettant de lire un flux d'entrée depuis le *terminal*. C'est grâce à lui qu'on peut interagir avec l'utilisateur uniquement par texte.

Le mode *terminal* possède un unique *Scanner* qui permet de lire le flux d'entré de l'utilisateur. Cependant leur utilisation peuvent poser problème :

- Lorsqu'une vague est terminé alors que le *Scanner* demandaient à l'utilisateur s'il voulait placer une tour, il n'est pas possible de fermer le Scanner de le l'ouvrir.

Malgré ce problème, il reste indispensable pour un fonctionnement correct du mode *terminal*.

2.1.2) Les Coordinates :

> La classe **Coordinates** est l'une des classes primordiale pour le bon fonctionnement du jeu. C'est grâce à elle qu'est gérer l'emplacement des objets des classes `Tower`, `Mob` et `Projectile`.

Comme dit précédemment, elle permet de gérer l'emplacement des objets des 3 classes d'`Entity`, et donc de gérer leur vitesse et leur emplacement. Elle contient plusieurs attributs important :

- Les attributs `x` et `y` qui sont de type *float* et permettent d'avoir une précision presque parfaite de l'emplacement des *entités* dans l'écran.
- Les attributs `speed` et `speedMax` qui sont de type *float* et permette de faire bouger les entités en fonction de ce paramètre.

Pour ce qui est des déplacements, on retrouve 2 méthodes permettant des déplacements de gauche à droite :

- La méthode `moveLeft()`, très utile pour les *Mobs*, permet de faire déplacer une *entité* de gauche à droite en fonction de sa vitesse.
- La méthode `moveRight()`, très utile pour les *Projectiles*, permet de faire déplacer une *entité* de droite à gauche en fonction de sa vitesse.

Pour le mode *terminal*, bien que les fonctions précédentes sont aussi utiles, on retrouve une méthode spécifique pour ce mode :

- La méthode `coordinateToPoint(String c)` prenant en argument un *String* et permet de transformer un String en coordonnées. Par exemple le *String* “A1” représente les coordonnées (0,1). La méthode est très utile pour placer facilement et intuitivement les *towers*.

  ⛔ Bien qu’elle puisse faire penser à des coordonnées en Mathématiques, ce n’est pas le cas car pour les coordonnées de type (x,y) :

  - **x** représente la hauteur en fonction de la Map
  - **y** représente la largeur en fonction de la Map

2.2.3) L’implémentation de Model et de Map :

> La classe **Model** est la classe la plus importante du projet. C’est elle qui gère le fonctionnement du jeu et les modifications faites par l’utilisateur.

Elle implémente à l’intérieur de sa classe la logique permettant de faire tourner le jeu. La logique est faites grâce à la fonction :

- `startWave()` qui gère toutes les vagues du jeu grâce à 3 *timer* primordial :
  - `projectileRound()` qui s’occupe du déroulement de chaque *Projectiles*. Normalement, le premier *projectile* traité sera toujours celui qui se trouve devant en premier. Lorsque le projectile touche une cible, elle s’ajoute dans une `ArrayList` qui s’occupera de supprimer tous les projectiles mort. Si elle tue un *Mob*, elle va arrêter son Timer lui permettant d’attaquer. Sinon, si

aucune tour est directement à côté du *projectile*, elle s'occupe de faire avancer le projectile.

- `towerRound()` qui s'occupe du déroulement de chaque *Towers*. Il s'occupera de lancer le `Timer` des *towers* si un *mobs* est présent dans sa trajectoire. Sinon, il arrêtera leur `Timer` si aucun entité n'est dans la trajectoire. C'est grâce à lui que les projectiles sont créer et sont ajoutés dans le plateau de jeu.
- `mobRound()` qui s'occupe du déroulement de chaque *Mobs*. Normalement, le premier *mob* traité sera celui qui est le plus proche du centre en premier. Il s'occupe d'enlever les tours tué par le *mob* actuel et lancera le `Timer` du *mob* si une tour est à proximité. Sinon, elle arrête le `Timer` si besoin et fera avancé le *mob*.

Elle contient aussi des méthodes tels que `moveAsMob()` et `moveAsProjectile()` qui gère le déplacement des entités et pour le mode *terminal* leur placement.

> La classe **Map** est une classe importante, avec la classe **Cell**, pour le bon fonctionnement de `Model`. C'est elle qui représente le plateau de jeu avec une matrice rempli de **Cell** et permet donc un placement des *tours, entités* et *projectiles*.

C'est dans la classe `Map` qu'est vérifié si un `Entity` est bien placé dans une *cellule*. Notamment grâce à la méthode `isValid(Coordinates c)` qui permet de savoir si les coordonnées donné en argument sont valide.

2.2.4) La classe Style :

> La classe **Style** est une classe principalement pensé pour le mode *terminal* permettant de customiser généralement le jeu en ayant toutes ses méthodes statiques et donc accessible à l'ensemble des classes.

Elle reste important pour le mode *terminal* car c'est elle qui envoie les messages d'erreur et la gestion des couleurs pour le mode terminal.

2.3) Le mode Graphique :

> Le **mode Graphique**, est le mode permettant d'utiliser une interface graphique pour pouvoir jouer au jeu. Elle partage le même but que le mode *Terminal* mais elle permet une interaction graphique, et non une interaction textuelle, avec le jeu.

Le mode *graphique* est développé directement après le mode *terminal* qui a permis un développement rapide du mode *graphique*.

2.2.1) Mise en place d'une machine à états :

> La classe **Game** est la classe principale. C'est ici que le choix du mode *graphique* ou *terminal* sera fait :
>
> - Si la commande lancé contient à la fin `--terminal` ou `-terminal`, alors le mode *terminal* sera lancé
> - Sinon, le mode *graphique* sera lancé.

Lorsque le mode *graphique* est lancé, le jeu en général, chaque état sera instancié sous le principe d'une machine à états singleton.

> Une **machine à état singleton** est une instance unique d'une machine à état unique dans le programme. Donc cela garantit que chaque classe représentant une vue n'a qu'une seule instance. Dans notre cas, l'utilisation d'une machine à état singleton ne respecte pas réellement le principe mais permet une mobilité plus simplifier.

Lorsque le programme est lancé, un seul `Frame` est utilisé, le *GameFrame*. Il s'occupera de changer le `JPanel` courant pour avoir un changement d'état fluide et donc ne pas besoin d'ouvrir une nouvelle fenêtre à chaque changement de vue.

Chaque vue vont implémenter l'interface `State` et chaque classe implémentant cette interface devront définir ces méthode :

- La méthode `enterState()` qui permet de gérer le cas où l'ancienne vue change pour la nouvelle vue et donc permet de charger toutes les components et méthodes nécessaire pour le bon fonctionnement de la vue.

- La méthode `quitState()` qui gère le cas d'un changement de vue. Elle permet généralement de stocker les variables modifié par l'utilisateur, comme par exemple avec un `JSliders` en prenant sa valeur, et de les partager à la classe `Model`.
- La méthode `getView()` qui permet de récupérer l'unique instance et donc de l'afficher dans le *GameFrame*.
- La méthode `refresh()` qui permet d'actualiser l'affichage de la vue actuelle, utile lorsqu'on utilise des éléments graphique.
- La méthode `isFirstTime()`, de pair avec `notFirstTime()`, permet de savoir si c'est la première fois que l'utilisateur entre dans la vue souhaité. Si c'est le cas, alors il devrait exécuter `enterState()` et ensuite exécuter `notFirstTime()`, sinon ne fait rien.
- La méthode `notFirstTime()`, de pair avec `isFirstTime`, permet d'indiquer à l'instance que la vue à déjà été visité.

Chaque instance est géré dans une classe `GameState` qui contient les instances suivantes :

- L'instance `MENU`, qui sera la première lancé et qui représente l'écran du début avec les choix : Jouer, Save et Quitter.
- L'instance `VERSION`, qui représente la vue permettant de sélectionner le mode *Normal* et le mode *Marathon*.
- L'instance `MAP`, qui représente la vue permettant de sélectionner l'une des 2 maps jouable.
- L'instance `PLAYING`, qui représente la vue où le plateau de jeu est présent et qui lance le jeu.
- L'instance `PAUSE`, qui représente la vue de Pause lorsque le bouton *Pause* de l'instance `PLAYING` est cliqué.

2.2.2) Les vues et leurs contrôleurs :

Les vues et les contrôleurs sont des éléments clés du mode graphique du jeu. Ils permettent d'interagir avec l'utilisateur et d'afficher les différentes interfaces graphiques.

- La vue `ScreenMenu` représente l'écran d'accueil du jeu, où les joueurs peuvent choisir de jouer, sauvegarder ou quitter le jeu. Elle est gérée par le contrôleur

`ScreenMenuController`, qui gère les actions de l'utilisateur et les interactions avec la vue.

- La vue `ChooseVersion` permet aux joueurs de sélectionner le mode de jeu, soit le mode normal, soit le mode marathon. Elle est gérée par le contrôleur `ChooseVersionController`, qui gère les actions de l'utilisateur et les interactions avec la vue.
- La vue `ChooseMap` permet aux joueurs de sélectionner l'une des deux cartes disponibles. Elle est gérée par le contrôleur `ChooseMapController`, qui gère les actions de l'utilisateur et les interactions avec la vue.
- La vue `Playing` est l'interface principale du jeu, où le plateau de jeu est affiché et où les joueurs peuvent jouer. Elle est gérée par le contrôleur `PlayingController`, qui gère les actions de l'utilisateur, les interactions avec la vue et la logique du jeu.
- La vue `Pause` est affichée lorsque le joueur met le jeu en pause. Elle est gérée par le contrôleur `PauseController`, qui gère les actions de l'utilisateur et les interactions avec la vue.

Chaque vue possède ses propres fonctionnalités et interfaces graphiques, permettant aux joueurs d'interagir avec le jeu de manière intuitive et agréable.

Elles héritent tous de la classe `Controller` et partagent tous la même méthode :

- `changeView(GameState state)` qui comme son nom l'indique permet de changer la vue entre une vue avec la vue donné en argument.

2.2.3) L'animation et la gestion des mouvements :

Pour ce qui est de l'animation, nous avons un *package Model* contenant des qui permet de gérer plusieurs aspect de la vue généralement utilisé par `Playing` :

- La classe `PlayingPanel` qui gère le placement des `Entity` dans le plateau de jeu en faisant des calculs de coordonnées et en actualisant les images utilisés pour l'entité.
- La classe `MapCell` qui permet une mise en place rapide et facile d'une map en donnant seulement en argument un *String* et qui le placera au bon endroit.

4) Conclusion :

Malgré le fait qu'on ai pas pu réaliser l'entièreté de notre objectif, nous sommes assez content du résultat en prenant en compte du manque de temps dû aux

examens et grâce à cela, nous avons gagné des connaissances en programmation et en interface graphique.

L'animation des archers étaient l'un de nos but mais par manque de temps et de connaissance au niveau de l'animation, nous avons seulement réussi à faire tenir debout un archer avec une animation.

4.1) Étapes accomplis :

- Pouvoir réaliser des animations entre chaque scène pour donner l'air au jeu d'être plus fluide.
- Pouvoir avoir au moins 2 maps différentes.
- Avoir un jeu devenant plus difficile à la demande et aussi au fil du temps.
- Avoir une bonne connaissance de Swing et AWT pour le prochain semestre.
- Avoir une bonne maitrise de Git.
- Apprendre à travailler en équipe et trouver des compromis lors de conflits.

4.2) Étapes échoués :

- Nous voulions faire des animations entre les `JPanel` mais lorsque nous avons compris les limites de Swing et de AWT, nous avons pensez que cela n'était pas possible et surtout pas primordiale pour le moment.
- Nous voulions créer des objets spéciaux ayant des effets comme le gèle, la brulure, etc..
- Changement l'asset des entités en fonction de la map.
- On peut sauvegarder mais pas charger une partie