Projet Tower Défense 2019-2020

11 MAI

Julia ARNOUX
Kevin DANG
Maxime DRESLER
Sarobidy RAPETERAMANANA

I) Modélisation

1. Organisation du groupe

Le dépôt git a été créé très rapidement une fois que les informations concernant le déroulement de la matière ont été disponibles, comme nous savions déjà que nous allions travailler ensemble, afin d'avoir un environnement de travail prêt. Nous avons ensuite mis en place nos canaux de communication et comment nous allions travailler en suivant le processus scrum. Nous avons donc choisi de faire un sprint par semaine en utilisant la réunion avec le professeur encadrant comme réunion de groupe. Dès le début, nous avons donc réfléchi ensemble à la modélisation du le projet. Celle-ci a émergé assez rapidement avec à sa base, une séparation des données brutes de l'affichage.

2. Vue d'ensemble

Afin de suivre cette modélisation, nous avons directement créer différents packages afin de séparer le modèle de la vue. Ainsi par exemple, le package Datamodel contient toutes les classes concernant les objets qui vont être utilisé dans le jeu comme les ennemies et les tours tandis que le package TowerDefenseVisual concerne l'affichage. Le lancement du jeu est géré par la classe présente dans le package Launcher. Evidemment, tout le projet tourne autour de l'héritage entre les classes. Nous avons dans un premier temps réfléchi aux algorithmes principaux qui seraient au cœur du fonctionnement de notre jeu puis avons commencé à mettre en place l'implémentation petit à petit, en partant des fonctionnalités les plus générales jusqu'au plus spécifiques. Il est possible de se faire une idée plus globale et précise de la modélisation avec le diagramme de classes UML disponible sur le dépôt Git.

II) Implémentation

1. Hitbox

UNE IMPLÉMENTATION RÉUTILISABLE

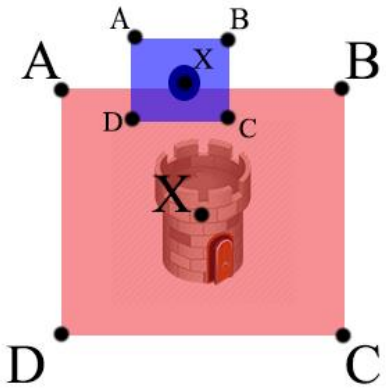
En commençant à réfléchir à la base de l'architecture de notre projet, l'un des principaux objectifs qui nous ont été donné et que nous gardions en tête, est d'avoir une base qui soit exploitable pour tous les autres projets de type collision. A savoir des Hitbox, permettant de gérer le contact entre différents éléments. Ici, entre, projectiles, bâtiments, et ennemis. Nous nous sommes donc décidés sur une class abstraite Hitbox

destinée à gérer les collisions. Tous les objets pouvant entrer en collisions (Projectiles, Ennemis, etc.) appartiennent donc à une classe héritant de Hitbox. La classe Hitbox a donc deux fonctions principales :

Premièrement, permettre de détecter lorsque deux objets Hitbox sont en collisions. Deuxièmement, déclencher la conséquence de cette collision entre les deux objets, grâce à une méthode abstraite, devant être redéfinie dans toutes les classes héritant de Hitbox afin d'avoir un comportement différent selon les types d'objets qui entrent en collision.

CONCEPT DÉTAILLÉ

L'implémentation de la classe abstrait Hitbox permet non seulement de gérer et détecter les collisions entre objet mais également de gérer leur position à tout moment. Voici un schéma qui permet d'en résumer rapidement le fonctionnement.



Chaque objet de type Hitbox possède cinq attributs principaux. Il s'agit de cinq points possédant s donc chacun des coordonnées x et y correspondant aux quatre coins du rectangle représentant la Hitbox ainsi que de son centre. Lorsque l'on cherche à déterminer si deux objets Hitbox sont en collision, l'on va croiser leurs coordonnées afin de voir si leurs Hitbox sont soit en contact, soient se superposent par endroit. Quant au centre, il permet de gérer les déplacements des objets de manière optimale grâce à la fonction updatePosition. En effet, une fois l'objet créé, plutôt que de déplacer chacun des cinq points de sa Hitbox à chaque mouvement dans le plan, il suffit d'en déplacer le centre. La fonction update Position mettra ensuite à jour toutes les autres coordonnées en prenant en compte la largeur et la hauteur de l'objet également stocké comme attributs de l'objet. Cette implémentation simple mais efficace nous permet de gérer à la fois les collisions et les déplacements de tout objet de type Hitbox et a représenté la base de notre travail

2. Placement des tours

CHOIX ET CONCEPT

On peut rapidement différencier deux types d'actions qui ont lieu sur l'interface graphique lors d'une partie. Premièrement, celles sur lequel le joueur n'a aucun contrôle, à savoir, le tir des tours, qui est automatiques, ainsi que le déplacement des ennemis, qui s'effectuent donc tous en tâche de fond. La manière dont le joueur peut interagir avec le jeu est la suivante : Placer des tours sur le niveau. Tout le mécanisme de déplacement des tours est donc implémenté dans les event listener des tours. Puis qu'un niveau est divisé en Tiles, dans un quadrillage clairement visible à l'écran, nous avons fait le choix de ne permettre au joueur de poser des tours uniquement sur un groupe de Tile correspondant à la Taille de la tour et suivant le quadrillage. Ainsi pour que le joueur n'ait pas de difficulté à bouger la sourie jusqu'à trouver une position valide, nous avons fait en sorte que lorsque le joueur bouge sa souris, la tour ne puisse s'arrêter que sur des positions valides.

IMPLEMENTATION

Lorsqu'une tour est créée, elle est ajoutée dans un des emplacements de la boutique. A ce moment-là, elle n'est pas encore ajoutée dans le niveau. Lorsque le joueur clique sur une tour, on change alors le parent auquel appartient son nœud graphique, pour la déplacer de la boutique, au niveau en lui-même. Une fois ceci fait, tant que le joueur ne lâche pas sa souris, il est libre de déplacer sa tour sur le niveau. Pendant tout ce temps, seule la position de la représentation graphique de la tour est mise à jour, met la position de sa hitbox, elle, ne bouge pas. Ce n'est qu'une fois que le joueur relâche sa souris que le processus de placement commence à opérer. On va alors vérifier que la position de la tour est valide. Vérifier qu'elle suit bien le quadrillage, qu'elle ne se trouve pas en dehors des limites du niveau, qu'elle ne se trouve pas sur le chemin ou sur une autre tour.

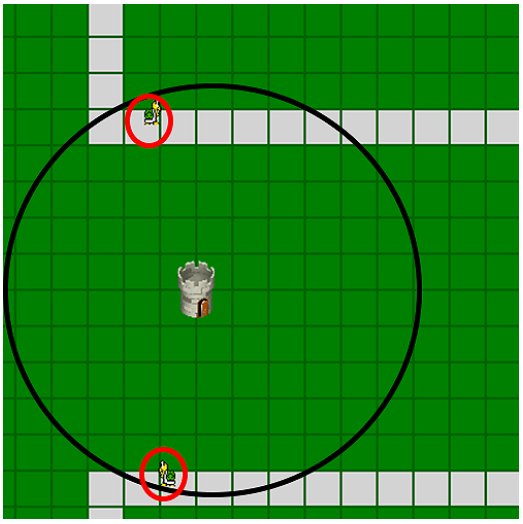
Si la position choisie par le joueur est valide, la tour sera alors placée sur le niveau de manière définitive, et sa Hitbox sera déplacée. La tour sera alors activée et pourra

commencer à viser et tirer sur des ennemis. Et une nouvelle tour d'un type aléatoire viendra alors remplir la place qu'elle a laissée vide dans la boutique.

En revanche, si la position n'est pas valide, on annule toute l'opération pour remettre la tour à sa place dans la boutique.

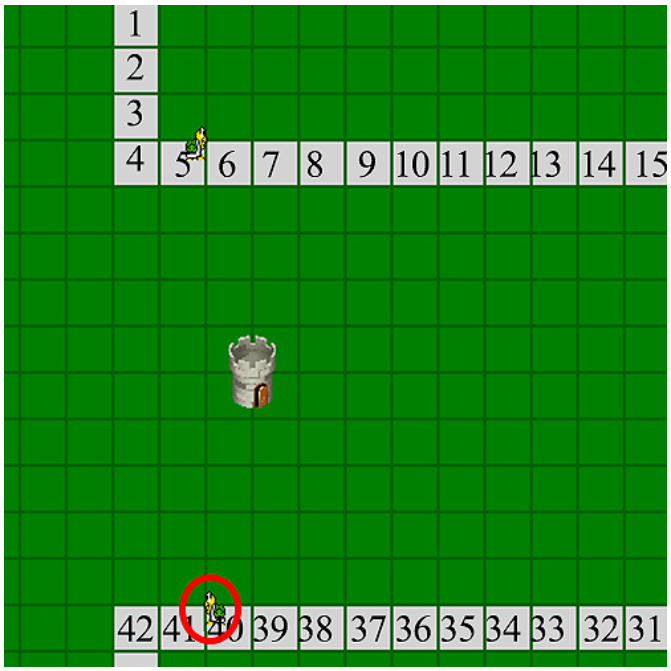
3. Algorithme de visée

ETAPE 1 : PORTÉE



Lors de la première étape de son fonctionnement, la première chose que l'algorithme de visée va faire, est de parcourir les ennemis présents sur le niveau (stockés dans une ArrayList) afin de ne sélectionner que ceux qui se trouvent à sa portée, comme vous pouvez le voir dans le schéma fourni à côté.

ETAPE 2 : DISTANCE A LA FIN DU CHEMIN



Une fois la première étape effectuée, l'algorithme va procéder à une seconde sélection parmi les ennemis qu'il a déjà sélectionnés à l'étape une. Cette fois ci, il va choisir l'ennemi qui est le plus proche d'arriver au bout du chemin jusqu'au château, et donc d'infliger des dégâts au joueur. Cela est rendu possible par l'indice des tiles. Chaque Tile qui n'est pas du chemin possède l'indice 0, l'indice des tiles de chemin en revanche, correspond à une numérotation partant de 1 pour le point de départ du chemin où apparaissent les ennemis, jusqu'au château. Chaque ennemi possède un attribut indiquant la Tile ou il se trouve actuellement, cette variable étant mise à jour dès que nécessaire lorsqu'il se déplace (Voir section sur le déplacement des ennemis), il suffit alors de sélectionner

l'ennemi se trouvant sur la Tile avec le numéro le plus élevé. Une telle étape permet d'avoir des niveaux avec des chemins faisant des détours et des boucles tout en ayant des tours qui visent l'ennemi le plus dangereux actuellement à sa portée. (Dans le sens le plus proche de la fin du chemin). Dans le cas ou deux ennemis se trouveraient sur une même Tile remplissant le critère, on les départagera alors en choisissant l'ennemi qui est le plus proche de la tour en distance cartésienne.

4. Déplacement des ennemies

CHOIX ET CONCEPT

Lors de nos discussions nous avons réfléchis à plusieurs moyens de déplacements des ennemis et la question qui revenait régulièrement était : comment l'ennemi saura trouver la case suivante ? Ainsi nos réflexions se sont axées sur cette question. Et donc nous avons pensé à un système simple qui indique l'ennemie si la Tile suivante se trouve au nord, à l'est, au sud ou à l'ouest de sa case.

IMPLEMENTATION

Pour chaque Tile, on lui fournit un tableau de boolean de taille 4 (un boolean pour chaque direction), un boolean true signifie que la Tile suivante du chemin se trouve dans cette direction. Ce tableau est mis à jour dans la méthode path() de Level, qui va permettre à toutes Tiles du chemin (path). Et on a la methode move() dans Enemy qui permet le déplacement de Enemy en utilisant la Tile courante de l'ennemi qui va nous indiquer la Tile suivante. Ainsi l'ennemi se déplacera selon sa vitesse prédéfinie.

5. Multi threading

DIFFICULTÉS

Après avoir mis en place un premier prototype permettant de déplacer les tours sur le niveau et de gérer le déplacement d'un unique ennemi et de faire tirer des projectiles il est rapidement apparu qu'afin de pouvoir afficher et calculer les positions et déplacements de multiples ennemis et projectiles ainsi que leur affichage il allait falloir utiliser plusieurs Threads. N'ayant jamais utilisé ce type de fonctionnalité auparavant cette implémentation c'est donc fait en tâtonnant. Le résultat pas forcément des plus optimal dans l'implémentation des threads le démontre mais grâce à l'expérience acquise et au conseils fournis par notre responsable de projet - que nous n'avons

malheureusement pas eu le temps d'appliquer car il nous aurait fallu reprendre de zéro une trop grande partie du code pour le temps qu'il nous restait-nous avons acquis de nombreuses connaissances à ce sujet.

IMPLEMENTATION

Afin que chaque objet de type tour, ennemi et projectile nécessitant une boucle d'action devant être répétée puisse être géré indépendamment des autres pour que les mises à jour des coordonnées et autres variables soient synchronisées nous avons ajouté à la classe Hitbox de laquelle héritent toutes ces classes, un attribut Thread, ou est stocké le Thread gérant chaque objet. Les instructions étant différentes selon sa classe. Nous avons également implémenté une classe Control SubThread permettant de démarrer et interrompre les Thread de manière plus simple et sécurisée.

CENTRALISATION ET INTERFACE GRAPHIQUE

Une fois la gestion individuelle et indépendante des objets de classe héritant de Hitbox mise en place, l'objectif que nous avons cherché à remplir a été le suivant, synchroniser ces threads et gérer le rafraîchissement de l'affichage graphique sur l'interface. Pour cela nous avons mis en place deux thread principaux. Le premier est dédié à la mise à jour des éléments graphiques. Toutes les mise à jour graphique peu importe depuis quel Thread elles sont demandées, sont toutes gérées par ce thread. Enfin, et non des moindres, le second Thread est celui qui centralise tout le fonctionnement d'un niveau. Ce thread principal permet de simuler le temps en jeu en calculant le delta time (temps qui s'est écoulé depuis la dernière frame) pour les frames mais également les ticks. A chaque Frame, ce thread principal s'assure que le Thread de l'interface graphique fait son travail puis le met en pause jusqu'à la frame suivant. Il fait de même à chaque tick avec les multiples threads des objets de Type Hitbox, s'assurant qu'ils font leur travail mettant ainsi à jour les données/position en jeu puis les mettant en pause jusqu'au prochain tick.

6. Design pattern

Sous les conseils du professeur encadrant nous avons étudié le design pattern (ou patron de conception) factory (ou fabrique). Celle-ci permettant de séparer la création

d'objets dérivant d'une classe mère de leur utilisation. De ce fait, on a alors la possibilité de créer plusieurs objets issus d'une même classe mère.

Après avoir étudié ce nouveau design pattern, nous avons décidé d'uniquement de nous en inspirer, pas de le reprendre complètement car nous avons dû faire des choix et accélérer le projet. Nous l'avons ainsi adapté afin de répondre à notre besoin.

Notre besoin étant qu'il fallait pouvoir construire plusieurs objets d'une même classe mais dont la valeur des attributs diffère en fonction du paramètre reçu.

Ainsi, la classe Tower possèdent une méthode static fabriqueTower() qui prend en paramètre un entier et fait appel au constructeur de Tower et renvoie ainsi une instance de Tower dont les attributs sont initialisés à de certaines valeurs prédéfinies en fonction de l'entier reçu en paramètre.

Nous avons répété ce design pattern pour les classes Wave, Enemy, Level.
Nous avons ainsi privilégié cette méthode pour ces classes-là afin d'ajouter plus facilement des nouveaux types de tours, de nouveau vague d'ennemi, des nouveaux types d'ennemis ou des nouveaux niveaux. En effet il suffit alors simplement de rajouter de nouveau cas supplémentaire dans les méthodes qui font appel aux constructeurs de la classes respectives pour fabriquer les objets.

7. Système de sauvegarde

CHOIX ET CONCEPT

On souhaitait avoir un système de sauvegarde dans notre jeu afin que l'utilisateur n'ait pas à recommencer le jeu à chaque fois qu'il arrête de jouer. De plus, il peut également supprimer cette sauvegarde pour recommencer. On parlera d'une réinitialisation de la sauvegarde, une nouvelle partie. Cette sauvegarde nous permet de mémoriser les niveaux réussit par le l'utilisateur.

IMPLEMENTATION

Ce système de sauvegarde fonctionne avec un fichier JSON, nous avons implémenté une classe Save qui a pour rôle de gérer ce système. Il y a donc une méthode qui nous permet de créer un fichier JSON avec la particularité de réécrire le contenue du fichier s'il existe. Une méthode qui permet lire le fichier JSON, cette méthode nous permet de récupérer le contenue du fichier. Puis nous une méthode pour décoder le contenue du fichier et le traduire un « objet simple de JAVA » (un tableau de boolean).

Enfin nous avons des méthodes qui vont permettre l'interaction entre le fichier JSON et les classes du jeu, comme la méthode sauvegarde() qui permet retourner les informations contenues dans le fichier JSON par les méthodes citées précédemment. Puis il y a la méthode miseAjour() qui permet de mettre à jour le fichier JSON grâce à la méthode qui crée un fichier JSON car cela va réécrire le fichier JSON avec les nouvelles valeurs. Et une méthode qui réinitialise le fichier JSON avec les valeurs par défaut.

De plus, à chaque lancement du jeu, un objet Save est créé. Alors nous vérifions si le fichier JSON existe afin de ne pas effacer la sauvegarde.

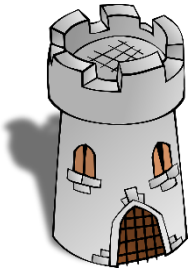
8. Les types de tours

Tour 1 :

Coût : 100
Dégâts : 20
Précision : 0.9
Portée : 150

Tour 2 :

Coût : 300
Dégâts : 80
Précision : 0.5
Portée : 200



Tour 3 :

Coût : 600
Dégâts : 70
Précision : 0.7
Portée : 300

9. Les types d’ennemis

Ennemie simple :

Point de vie : 25
Dégâts : 1
Vitesse : 1.5
Armure : 8

Ennemie moyen :

Point de vie : 50
Dégâts : 1
Vitesse : 1.5
Armure : 10

Boss :

Point de vie : 150
Dégâts : 2
Vitesse : 1.2
Armure : 15

Big Boss :

Point de vie : 1500
Dégâts : 4
Vitesse : 0.9
Armure : 50

Big Boss Final :

Point de vie : 10 000
Dégâts : 5
Vitesse : 0.7
Armure : 50

III) Difficultés

1. Difficultés techniques

Outre les problèmes de développement, nous avons eu beaucoup de soucis au niveau des technologies utilisés. Cela dû à la diversité des configurations de nos machines. Une difficulté majeure a été la compilation et l'exécution. Au départ le projet était uniquement exécutable sur nos IDE qui s'occupait également de la compilation. Pour répondre à la demande de produire quelque chose qui pouvait être compilé en ligne de commande nous avons choisi d'utiliser Gradle. Le problème étant que le projet arrivait à son terme et n'avait pas réellement l'architecture d'un projet Gradle. La prise en main de ce moteur de production a donc été très difficile. Nous voulions également produire un jar exécutable pour que le projet puisse être testé sans savoir que la librairie JavaFX que nous utilisons n'étais plus inclus dans la version 11 du JDK de Java ce qui a été source de confusion.

2. Coronavirus et confinement

Pour les membres du groupe, il est devenu difficile de travailler sur le projet pendant la durée du confinement pour diverses raisons. Le projet a donc été fortement ralenti et le produit final ne contient pas toutes les fonctionnalités que nous aurions souhaité implémenter. Cependant chacun a fait de son mieux pour apporter sa meilleure contribution possible au projet pour avoir un résultat fonctionnel même si nous n'avons malheureusement pas pu optimiser l'intégralité de notre code comme nous l'aurions souhaité.

IV) Bilan

Nous sommes satisfaits du résultat de notre travail, s'il n'a pas été exempt d'erreurs et de tâtonnement lors de l'utilisation de certaines fonctionnalités de java que nous venions de découvrir alors pour la première fois et que le rendu final n'est pas aussi complet que nous le voulions, nous estimons être arrivés à quelque chose de fonctionnel. Nous en avons également tiré des enseignements sur la gestion et l'organisation d'un travail en groupe et dans un temps limité, la répartition des tâches, une utilisation optimisée de

Git, et avons appris à utiliser de nombreuses fonctionnalités et bibliothèques de java (Threads, JavaFX, Json, Gradle, ...).