Rapport du projet de programmation orientée objet:

Jeu de stratégie Wargame

BERNARD Paul-Antoine, AIT ADDI Marwan, HUBERT Gustav

Janvier 2021

Table des matières

1 Présentation

Dans le cadre de notre 3e année de Licence d'informatique, nous avons réalisé un projet visant à programmer un jeu de stratégie de guerre en langage Java. Ce jeu consiste en l'affrontement de deux armées, l'une contrôlée par l'IA et l'autre par l'utilisateur. La partie est composée de tours de jeu durant lesquels seule une armée peut effectuer des actions. C'est à dire avancer ses troupes et attaquer les unités ennemies (si les contraintes le permettent). Le jeu se termine lorsque l'une des deux armées est anéantie.

1.1 Utilisation

Le jeu fonctionne de la manière suivante : lors d'une partie, le joueur peut déplacer ses troupes en cliquant dans un premier temps sur un soldat, puis dans un deuxième temps en cliquant sur la case adjacente voulue. De la même manière le joueur peut attaquer des monstres, qu'il soient disposés sur des cases adjacentes ou non (en fonction des types de soldats). Lorsque le joueur a fini de réaliser ses actions, il suffit d'appuyer sur le bouton "fin du tour". Les soldats qui n'ont pas encore joués se reposent alors et gagnent un peu de vie (s'ils ne se trouvent pas dans un désert, auquel cas ils en perdent). Les monstres jouent ensuite leur tour. La partie se termine lorsque l'un des deux camps ne possède plus aucun soldat. En passant la souris au dessus d'un soldat (ou en cliquant dessus en mode performance), plusieurs statistiques sont affichées, par exemple les points de vie. Le jeu peut être sauvegardé dans le menu, ouvrable avec le bouton en haut à gauche (ou en appuyant sur échap).

1.2 Techniques de POO mises en oeuvre

Nous avons appliqué dans le projet la majorité des techniques et conseils de POO que nous avons pu voir en cours :

- L'encapsulation : dans la pluspart de nos classes nous avons empêché la modification et la lecture des attributs depuis l'exterieur en créant des accesseurs et des mutateurs.
- Polymorphisme : en définissant dans plusieurs classes différents constructeurs pour nous permettre d'avoir de multiples options pour coder et simplifier certaines tâches.

1.3 Partage des tâches et du temps

Nous avons décidé dès le début du projet de séparer celui-ci en deux grandes parties, la partie affichage et la partie fonctionnement du jeu. Ce fonctionnement a été séparé lui même en deux, l'aspect unité avec les héros et les monstres et l'aspect plateau de jeu avec la carte, les positions. Chaque membre du groupe a réalisé des méthodes et écrit des morceaux de code dans différentes parties, mais le partage global du travail s'est réalisé de la manière suivante : Gustav s'est occupé de l'affichage, des menus ainsi que des Éléments, Marwan à quand à lui majoritairement géré les héros, les monstres ainsi que d'autres fonctions de positions et tests les concernant, Paul-Antoine s'est occupé de la carte, et de fonctions de positions tels que la recherche du chemin le plus court.

2 Menus et interfaces utilisateur

La plupart des menus se ressemblent : quelques boutons sur un fond de couleur ou une image, éventuellement un logo. La classe MenuSimple prend ces éléments et s'occupe de leurs mise en page. Cela simplifie la création des menus nécessaires : un menu principal, un menu pour les options et un pour le chargement de sauvegardes. Malgré le fait qu'ils soient tous construits sur la même base, des différences existent.

2.1 MenuSimple

Cette classe permet de construire un JPanel (panneau) sur lequel sont arrangés les éléments donnés pour créer le menu souhaité. Un GridBagLayout est utilisé pour cela, permettant de regrouper les boutons au centre de la fenêtre à taille fixe. Tout cela ne se passe pas directement sur le panneau qu'étend MenuSimple, mais plutôt sur un deuxième panneau qui rempli le premier. Cela permet, lorsque on le souhaite, de transitionner vers un autre menu ou panneau. Le panneau tenant le menu actuel est simplement rendu invisible et le nouveau prend sa place. L'opération inverse, le retour au menu d'origine, est ainsi très rapide. De ce fait, chaque menu garde en mémoire son parent et peut lui demander de se réafficher.

2.2 MenuPrincipal

Cette classe étend MenuSimple et l'utilise tel que conçu. Ce menu permet de :

- Lancer un nouveau jeu
- Charger un jeu existant (voir MenuChargerJeu)
- Accéder aux options du jeu
- Quitter le programme

2.3 MenuOptions et Paramètres

Cette classe crée également un menu simple. Mais contrairement au menu précédent, les boutons sont chargés dynamiquement. Chaque bouton correspond à un paramètre défini dans IConfig. Le constructeur récupère alors la valeur du paramètre (si elle est définie) dans le fichier de configuration. Avec chaque clic sur un des boutons la prochaine valeur possible est chargée et affichée. Lors du retour au menu principal les nouvelles valeurs sont enregistrées dans le fichier de configuration et peuvent être récupérées à l'aide de la classe Parametres. Les paramètres modifiables sont :

- La taille de la fenêtre : petite (1/4 de l'écran), moyenne (1/2 de l'écran), grande (3/4 de l'écran) et plein écran
- La difficulté du jeu (nombre d'ennemis) : simple, moyenne, difficile
- Déplacement vertical : modifie légèrement l'apect visuel du plateau
- Mode performance (réduit l'utilisation en mémoire et en processeur du jeu) : On et Off

2.4 MenuChargerJeu

Ce menu étend également MenuSimple mais crée sa propre interface. Après avoir récupéré le nombre de sauvegardes dans le fichier de configuration, ces dernières sont chargées et placées dans un volet de défilement. Cette liste est composée de JPanels permettant d'afficher quelques informations sur chaque sauvegarde (date de la sauvegarde, une capture d'écran du jeu, le temps joué et le nombre de soldats restant). Ces "cartes" ont une fonction et apparence similaire aux boutons translucides (voir plus bas). Au clic, une sauvegarde reste sélectionnée, et l'utilisateur peut choisir de la charger ou de la supprimer.

2.5 Aspects visuels des composants Swing

Afin d'améliorer l'aspect visuel des composants fournis par Swing, les boutons, volet de défilement et label sont étendus. Avec un override des méthodes paintComponent on peut les rendre translucide (voir transparent) et utiliser des dégradés de couleurs. En dessinant soit-même le texte on peut utiliser l'anti-aliasage et des nouvelles polices. Ces dernières sont générées par la classe GenPolice, qui permet de générer une police en fonction du poids et de la taille voulue.

3 Création et gestion de la carte

3.1 Géneralités

L'une des premières problématiques lors de la réalisation d'un jeu de plateau est de savoir comment l'on va représenter justement le plateau. Ici nous avons décidé de créer une grille d'éléments (de type Enum). Chaque élément représente un terrain cela peut être une plaine, un désert, une forêt… Sous java cela se traduisant en un tableau d'éléments, à deux dimensions. Mais cette grille est un peu particulière, en effet nous avons décidé à l'affichage de représenter la carte non pas par une grille rectangulaire, mais par une grille hexagonale. Ce choix a donc affecté les fonctions de la classe ainsi que l'affichage. Il a fallu prendre en compte qu'avec une grille hexagonale, si l'on se trouve à une position (i,j), on ne peut pas se déplacer sur les 8 cases adjacentes. On ne peut ni se déplacer en haut à droite, ni en bas à droite : on obtient ces contraintes si l'on représente la grille hexagonale avec chaque hexagone aligné et non pas collé comme à l'affichage.

3.2 Ajout des Soldats

Nous avions donc désormais notre classe Carte.java qui sert de support aux autres fonctionnalités du jeu. Nous avons ensuite décidé de renseigner une grande partie des informations de jeu directement sur cette carte. Pour cela, nous avons créé des champs et des fonctions pour savoir si des soldats étaient présents sur un élément ou si celui-ci était libre.

3.3 Composition de la classe

La classe Carte.java est composée de fonctions d'initialisations, et de tests. On peut y retrouver des fonctions qui permettent à partir d'une position de trouver dans ses voisins un héros ou une position vide (sur laquelle on peut se déplacer). Il y a aussi la fonction de placement de zones de spawn où nos deux armées pourront apparaître, ainsi que la fonction de placement aléatoire des soldats dans ces zones de spawn.

4 Soldats et armées

4.1 Soldats en général

Les soldats (Héros ou Monstres) sont gérés majoritairement avec les fonctions de la classe abstraite Soldat, nous avons modifié plusieurs fonctions de base ainsi que les monstres et héros par défaut. Les soldats sont par exemple capable de déterminer eux-mêmes ce qui se passe lors d'un combat uniquement avec la methode combat(Soldat soldat). La plupart des fonctions ont très peu de prérequis et verifient ce qu'elles doivent faire toute seules ce qui facilite l'utilisation des classes filles (Heros et Monstre).

4.2 La methode actionHeros

Cette methode de la classe carte est l'une des fonctionnalités les plus importantes liée aux Heros, elle permet en lui donnant uniquement deux coordonées de déterminer l'action à faire. Sur notre interface graphique on clique juste sur une case où se situe un heros puis sur une autre case et la fonction s'occupe de déterminer ce qu'il y a à faire, soit on ne peut rien faire, soit on déplace le héros en question, soit on attaque, que ce soit = distance ou au corps-à-corps. Elle utilise également toutes les methodes liées aux Heros, telles que combat() ou deplacerSoldat().

4.3 Deplacement des Soldats

Le déplacement des soldat se fait en utilisant des fonctions de la classe Position, afin de tester tout d'abord si la case où l'on souhaite se déplacer est adjacente (estVoisine(Pos)) ou valide. Puis on va devoir tester si la case est libre, si elle est accessible (c'est à dire que le terrain n'est pas un obstacle) etc.

De par le plateau représenté par des hexagones et non des carrés, nous avons du prendre en compte quelques cas particuliers, par exemple la numerotation des cases adjacentes que l'on peut faire à partir d'une case n'est pas la même sur les lignes paires et impaires, nous avons donc trouvé la manière la plus concise de regler ce problème.

4.4 Calcul de la vision

Le calcul de la vision des soldats est effectué avec la fonction calculerVision() qui va appeler la fonction récursive (et privée) calcVisRec() qui prend en paramètre la portée et la vision puis va cacher ou découvrir les cases selon le paramètre boolean "cache". La fonction s'appelle récursivement sur les cases voisines en diminuant la portée de 1 à chaque fois, ainsi elle s'arrête quand la portée est inférieure a 1. Lorsqu'une case est appellée par cette fonction elle devient visible (ou cachée selon le paramètre) et ainsi on peut calculer rapidement et facilement la vision de tous les Héros.

5 Affichage du plateau de jeu

L'affichage s'effectue à l'aide de la classe PanneauJeu qui étend JPanel. Le constructeur de celle-ci prend la carte à afficher/modifier en argument. Le plateau de jeu est directement dessiné sur le panneau, selon l'emplacement et le zoom voulu. Ainsi l'utilisateur peut bouger le plateau en tirant dessus ou en utilisant son clavier (flèches, WASD ou ZQSD) et zoomer/dézoomer sur l'emplacement de son curseur à l'aide de la molette de souris ou de son clavier (Page Up et Page Down).

5.1 Création de l'interface

Pour éviter de tout recalculer avec chaque mouvemement, le plateau est dessiné dans une image qui fait guise de buffer et n'est mis à jour que lorsqu'il y a des changement. De plus, pour économiser la mémoire et le processeur, de plus petites images sont chargées, le plateau est dessiné sur fond noir et des algorithmes d'affichage plus rapides sont utilisés (par Swing/AWT). Quelques composants Swing (boutons, label) sont également utilisés et affichés après le plateau pour qu'ils apparaissent au dessus. Leurs placements est déterminé par un GridBagLayout.

5.2 Gestion des clics

Puisque le plateau n'est qu'une image, cette classe s'occupe également des clics sur celle-ci. Pour déterminer où, sur la carte, l'utilisateur a cliqué on utilise un tableau des hitbox à la taille du buffer plateau. En utilisant la position du clic comme indices dans ce tableau on peut retrouver les coordonnées de la case dans la carte. Muni de celles-ci on peut exécuter les actions telles que décrites dans la partie précédente. La même technique est également utilisée pour afficher des informations sur un soldat. On affiche alors les labels de la barre d'information avec les bonnes informations.

5.3 Dessin du plateau

La classe Carte s'occupe de créer le tableau des hitbox et l'image du plateau lors du premier affichage. Elle traverse les cases et détermine leurs emplacements, puis leur demande de s'afficher et de modifier le tableau des hitbox. Faire cela de haut en bas crée un effet de perspective, les cases plus proches cachant les cases derrière elles. Chaque case (définie par la classe Element) récupère l'image correspondant à son type et s'affiche au bon endroit. Une deuxième image légèrement plus foncée est utilisée lorsque aucun soldat n'est proche. Si une unité se trouve sur la case, elle est finalement affichée avec une petite barre indiquant ses points de vie. La mise à jour du tableau des hitbox n'est pas immédiatement nécessaire et est donc effectuée dans un thread parallèle. L'emplacement de chaque pixel de chaque case est utilisé dans le tableau des hitbox. Si le pixel n'est pas transparent on place les coordonnées de la case correspondante dans le tableau.

5.4 Affichage des dégâts

Pour mieux visualiser ce qui se passe, une animation (de deux images toutefois) est utilisée lorsqu'un soldat prend des dégâts. Une image teintée en rouge apparaît pendant une demi seconde à la place de l'image du soldat habituelle. Même s'il s'agit d'une courte durée, pauser l'exécution pendant une demi-seconde en attendant de réafficher la case n'est pas une option : toute l'interface serait bloquée. On lance donc un nouveau thread pour lequel il n'est pas important de toujours rester actif.

5.5 Rafraîchir le plateau

Redessiner le plateau entier avec chaque changement est simple mais très lent. Il est beaucoup plus efficace de redessiner chaque case au besoin. L'utilisation de la perspective rend cela, malheureusement, difficile. Une case pourrait avoir caché une autre au dessus d'elle (si elle avait un soldat par exemple). En plus de la case elle-même il faut donc redessiner les deux cases au dessus d'elle. Mais si ces trois cases sont elles-mêmes cachées par d'autres cases au dessous d'elles, il faut également les redessiner et ainsi de suite. Suivant l'emplacement de la case à réafficher originalement, il faudrait également réafficher une cascade d'autres cases au dessous d'elle (jusqu'à la bordure du plateau). Ce ne serait pas beaucoup plus efficace que de tout réafficher.

Au lieu de réafficher les cases au dessus et au dessous d'une autre en entier, on se limite aux parties qui pourrait être affectées par une case à redessiner (en rouge sur le schéma). C'est à dire les moitiés des éléments juste au dessus et des quarts juste au dessous de la case qui nous intéresse. Il ne suffit pas de redessiner l'image de la case dans ces parties puisqu'elle peut être dans plusieurs états. Chaque case possède donc sa propre image buffer qui est mise à jour avec chaque changement. Puisqu'elle n'est pas nécessaire immédiatement elle est également créée dans le thread parallèle utilisé pour le tableau des hitbox. Le buffer pourra ainsi être utilisé pour redessiner les parties affectées par la mise à jour d'une case adjacente. De plus, on met à jour le tableau des hitbox sur les parties affectées.

Ce système de rafraîchissement ajoute une nouvelle difficulté : lorsque plusieurs cases doivent être mises à jour en même temps, une case peut utiliser un buffer qui n'est plus actuel. Des quarts de soldats peuvent ainsi rester sur des cases autrement vide. Pour contrer cela, on utilise une file de cases à rafraîchir. On peut ainsi déterminer l'ordre d'actualisation de cases proches et minimiser ce problème. Il est néanmoins impossible d'exclure tous les cas.

6 Intelligence Artificielle

6.1 Recherche d'un chemin

6.1.1 Généralités

Lors de la conception du jeu, nous nous sommes demandés comment nous pourrions écrire une fonction qui indiquerait le chemin le plus rapide d'une position à une autre. Nous avons au début pensé à implémenter l'algorithme de Djikstra mais nous avons finalement opté pour l'algorithme A* qui est plus rapide à mettre en place et donc à tester et intégrer au projet.

6.1.2 Fonctionnement de l'algorithme A*

Cet algorithme fonctionne avec deux listes de nœuds. Les nœuds représentent ici des positions. On part de la position de départ, et on ajoute toutes les positions voisines accessibles dans une liste nomée liste ouverte. On sélectionne ensuite dans la liste ouverte le noeud le plus avantageux pour atteindre le point d'arrivée. On le place dans une liste appelée liste fermée. On réitère ce processus avec le noeud que l'on vient d'ajouter dans la liste fermée jusqu'à arriver trouver ou non le noeud de destination. Si il existe un chemin, on obtient à la fin de l'algorithme une liste de nœuds et donc de positions du point de départ au point d'arrivée.

6.1.3 Calcul des coûts

L'un des aspects les plus importants de l'algorithme A* est la manière de donner un coût à chaque noeud. Pour notre projet, nous avons choisi de nous servir des distances euclidiennes entre les différents points pour hiérarchiser les différents noeuds. Le coût total d'un noeud est le résultat de la somme de différents coûts. Pour un point (i,j) il y'a tout d'abord le coût de "départ" qui est la distance entre le point de départ et (i,j). Il y'a ensuite le coût "d'arrivée" qui est la distance entre (i,j) et le point d'arrivée. Le coût total est égal à la somme entre le coût de départ et le coup d'arrivée.

7 Gestion des sauvegardes

Des sauvegardes peuvent être créées avec une partie lancée puis plus tard relancées à partir du menu. Nous allons voir comment celles-ci sont créées et chargées.

7.1 Création

Lorsque le joueur souhaite sauvegarder sa partie, un objet de la classe GameSave est créé. Il contient quelques informations sur la partie en cours (décrit dans la partie 2.4). Cet objet récupère également la carte actuelle du jeu. Puisqu'on ne veut pas immédiatement charger toutes les cartes des sauvegardes dans le menu de chargement des jeux, celles-ci ne sont pas enregistrées dans GameSave. Lors de la création de la save, la carte est donc enregistrée séparément et seul le nom de cette sauvegarde est gardée. Finalement l'objet GameSave lui même peut être enregistré à l'aide de l'interface Serializable de Java.

7.2 Récupération

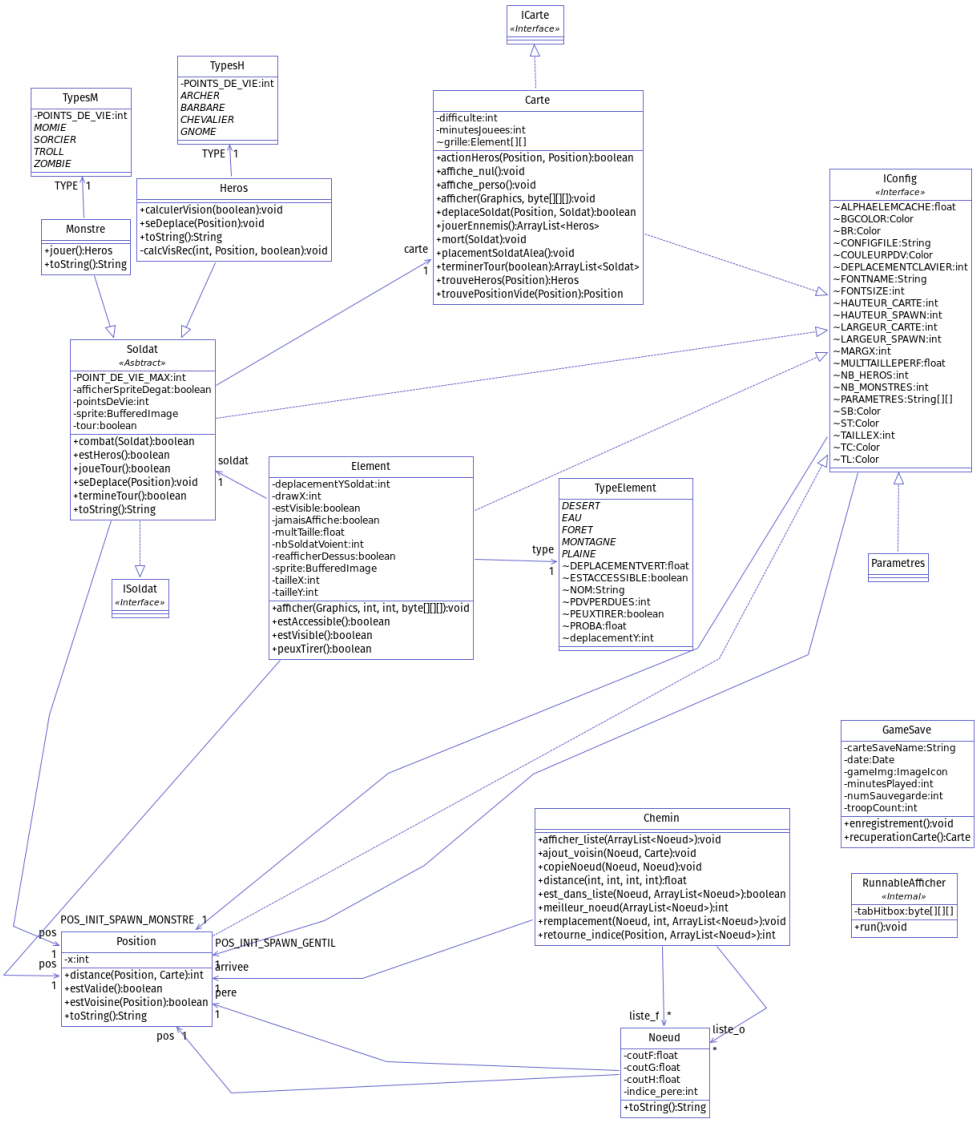
Pour retrouver les sauvegardes (enregistrées dans le répertoire courant du jeu), on tient un compte du nombre de sauvegardes dans le fichier de configuration. Toutes les sauvegardes étant numerotées, on peut ainsi les retrouver. Après avoir été affichées dans le menu décrit auparavant, et que l'utilisateur en ai choisit une, on récupère la carte à l'aide du nom sauvegardé. Avec celle-ci on crée un nouveau PanneauJeu (décrit dans la partie 5).

8 Conclusion

Pour conclure, nous sommes assez satisfaits du travail que nous avons fourni et du projet que nous rendons. Nous pensons avoir répondu aux principales problématiques posées tout en ajoutant nos propres idées et améliorations du jeu. En revanche nous aurions pu perfectionner notre jeu en ajoutant divers détails comme l'implémentation de différents types de soldats, des animations plus poussées lors de l'affichage ou encore consolider les bases du jeu en améliorant les caractéristiques des soldats et leurs interactions avec la carte par exemple.
Nous avons tous les trois trouvé le projet intéressant et utile. En effet celui-ci

montre la force d'un langage orienté objet comme Java en nous présentant les avantages de la hiérarchisation des classes et l'utilisation intuitive des objets. Nous avons pu nous rendre compte qu'il est très simple par exemple de faire intéragir des classes entre elles (dans notre cas les élements, la carte et les soldats par exemple).

9 Diagramme des classes



10 Liste des ressources utilisées

https ://fonts.google.com/specimen/Raleway
https ://flaticons.net
https ://assetstore.unity.com/packages/2d/environments/painted-2d-terrain-hexes-basic-set-52258
https ://craftpix.net/freebies/
https ://stackoverflow.com/a/7603815/5591299 (redimensionnement d'images rapide)
https ://stackoverflow.com/a/33286979/5591299 (JScrollPane customisé)
https ://stackoverflow.com/a/36744345/5591299 (teinter une image)
https ://docs.oracle.com/
https ://ateraimemo.com/Swing/TranslucentButton.html (base pour le bouton translucide)